Università degli Studi di Napoli "Federico II"

Facoltà di Scienze Matematiche Fisiche e Naturali

Tesi di Laurea in Scienze Informatiche

Analisi e Sviluppo di un prototipo di oggetti persistenti per il modello dei dati dell'esperimento CMS.

Relatore:
Dott. Luca Lista

Laureando:
Alessandro Esposito
matr. 50 / 406

Anno accademico
2002 – 2003

. . .a tutte le persone che hanno creduto in me.

“Solo gli imbecilli non hanno dubbi”
“Ne sei proprio sicuro?”
“Non ho alcun dubbio!”

IL DUBBIO, LUCIANO DE CRESCENZO, MONDADORI ED., 1992

Indice

Indice i

Ringraziamenti 1

1 Introduzione 2

2 L'esperimento CMS a LHC 4
2.1 Il rivelatore CMS 8
2.2 Struttura dell'apparato sperimentale 9
2.3 I sottorivelatori di CMS 11
2.3.1 Il Tracker 11
2.3.2 Il sistema calorimetrico 13
2.3.3 Il magnete 14
2.3.4 Il sistema di rivelazione dei muoni 14
2.3.5 Il sistema trigger 17

3 Il software di CMS **22**
3.1 Le componenti del software CMS 22
3.2 L'architettura software di CMS 24
3.2.1 I Framework in generale 25
3.2.2 Il Framework di CMS: COBRA 25
3.3 Schema delle attività principali dell'esperimento. 27
3.4 La simulazione 27
3.5 Il processo di Ricostruzione 31
3.5.1 La ricostruzione *"on demand"* 32

3.5.2 Le fasi della ricostruzione 34
3.5.3 Le classi principali utilizzate nel framework 36
3.6 I Jet 37
3.6.1 Algoritmi basati sui Coni 39
3.6.2 Algoritmi di tipo "Jade" 41

4 La persistenza degli oggetti per l'analisi. Studio di un caso particolare: i *"jet"* 43
4.1 Introduzione 43
4.2 Fase di Analisi 44
4.2.1 Prima iterazione sul *design* 45
4.2.2 Seconda iterazione sul *design* 51
4.2.3 Terza iterazione sul *design* 54
4.3 Fase di Sviluppo 56

5 Conclusioni 63

A Appendice A 66

Elenco delle figure 86

Bibliografia 88

Ringraziamenti

La presente tesi è il risultato di un lungo lavoro di profonda ricerca e attività, durante il quale ho avuto modo di essere affiancato e sostenuto da amici vecchi e nuovi, e da professionisti nei confronti dei quali la mia stima è andata via via crescendo. Approfitto, così, per consegnare i miei più sinceri ringraziamenti al Dott. Luca Lista, che senza remore mi ha consigliato e supportato in questa lavoro permettendomi di affacciarmi ad un nuovo mondo. Ringrazio profondamente anche il Dott. Francesco Fabozzi dell'Università della Basilicata senza il quale questo progetto non avrebbe avuto tutte le sfumature necessarie. Un ringraziamento particolare va alla mia ragazza, Pamela, che mi ha sempre incoraggiato, caricandomi di ottimismo anche nei momenti più difficili, ed inoltre, ha saputo pazientemente ascoltare anche i miei noiosi discorsi, privi di ogni logica per tutte le persone che come lei non hanno particolare interesse e conoscenza dell'informatica. Come non menzionare, infine, i miei genitori e i miei fratelli, che nonostante la tensione, mi sono sempre rimasti vicini, sostenendomi in qualunque modo. Posso quindi, essere pienamente soddisfatto del percorso svolto che ha posto in me delle forti basi su cui costruire il mio futuro e che credo, mi accompagnarà per la vita.

Capitolo 1

Introduzione

Il CERN (European Organization for Nuclear Research) è uno dei più grandi laboratori di ricerca nel mondo. Fondato nel 1954, è situato al confine tra Francia e Svizzera, poco fuori la città di Ginevra. Il laboratorio, finanziato da venti nazioni europee, vede la collaborazione di oltre settemila scienziati provenienti da tutto il mondo, che si prefiggono lo studio delle particelle costituenti la materia e della natura delle forze che le governano.

La complessità dei recenti esperimenti di Fisica subnucleare richiede l'uso di tecnologie avanzate e l'impiego costante di notevoli risorse umane, nonchè un cospicuo apporto di capitali. Il CERN è attualmente impegnato in una impresa ardimentosa: la realizzazione di un nuovo acceleratore, il Large Hadron Collider (LHC), dove fasci di protoni ad alta intensità verranno fatti collidere con un valore di energia mai raggiunto prima (fino a circa 14 TeV[1]). Queste condizioni estreme offriranno nuove possibilità di ricerca, quali la verifica di fenomeni previsti dalla teoria ma non ancora dimostrati sperimantalmente e la scoperta di nuovi costituenti della materia. Per compiere questi studi saranno posti quattro rivelatori sull'acceleratore LHC allo scopo di poter fornire il maggior numero di informazioni possibili sulla fisica che può essere prodotta nelle collisioni protone–protone. Il più grande dei quattro rivelatori utilizzati per questi studi sarà CMS (Compact Muon Solenoid), attualmente in fasi assemblaggio che sarà operativo a partire dal 2007. Pro-

[1] $10^{12} electron - Volt$

gettato per misurare con alta precisione le particelle prodotte dall'interazione protone–protone, è stato sviluppato per confermare la correttezza di alcune teorie (meccanismo di rottura della simmetria di Higgs) e di esplorare possibili fenomeni di nuova fisica (teorie di supersimmetria [12]) che potrebbero essere osservabili ad energie non accessibili agli acceleratori fino ad oggi costruiti. Gli studi degli scienziati si baseranno sulle informazioni riportate dai rivelatori di CMS (e dagli altri esperimenti) realizzati sull'acceleratore LHC; occore però precisare che le informazioni fornite dall'elettronica di lettura dei rivelatori sono in uno stato ancora troppo "grezzo" per poter consentire agli scienziati di effettuare il lavoro di "analisi", e completare quindi gli studi di Fisica. Questi dati forniti dall'elettronica costituiranno la base di partenza per poter ricostruire un modello dei dati più sofisticato e più adatto all'analisi fisica. In particolare, questo modello verrà realizzato grazie al supporto di un linguaggio di programmazione d'alto livello orientato agli oggetti (C++), ad un complesso framework implementato tramite tale linguaggio, e con l'utilizzo dei Design–Patterns. La maggior parte di questi oggetti ricostruiti dovrà essere memorizzata in maniera persistente su un dispositivo di massa; in tal modo tali oggetti, saranno disponibili nella fase di analisi off-line senza necessità di ripetere lunghi calcoli.

Mentre acceleratore (LHC) e esperimento (CMS) sono in fase di istallazione, già da tempo si lavora allo sviluppo del software di ricostruzione, che attualmente utilizza un'ulteriore software di simulazione dei dati dell'esperimento. Tra gli istituti che collaborano a CMS è presente anche la sezione di Napoli dell'Istituto Nazionale di Fisica Nucleare (I.N.F.N.), che in particolare si occupa della costruzione di alcune parti del rivelatore e dello sviluppo del software. Questo lavoro di tesi, svolto appunto in collaborazione con la sezione di Napoli dell'INFN, si prefigge come scopo la creazione di un prototipo di oggetti persistenti per il modello dei dati dell'analisi fisica, che costituisce un contributo al software ufficiale dell'esperimento. In particolare si cercherà di rendere persistente una particolare classe di oggetti chiamata jets a partire dalla struttura transiente esistente, ovvero da un'architettura in grado di ricostruire i jet, ma non di memorizzare tali oggetti sul disco. Questo prototipo fornisce soluzioni a problemi abbastanza generali nell'analisi dei dati, che potranno essere usate per altri tipi di oggetti previsti nel modello dei dati.

Capitolo 2

L'esperimento CMS a LHC

CMS (Compact Muons Solenoid) [1] è un esperimento di Fisica delle particelle elementari che operarà sul più grande acceleratore di particelle che sia stato mai progettato: LHC (Large Hadron Collider) [2] attualmente in fase di costruzione presso i laboratori del CERN (European Organisation for Nuclear Research) [3] di Ginevra. La portata dell'esperimento è talmente grande che attualmente vi collaborano 160 Istituzioni e circa 2000 tra scienziati ed ingegneri provenienti da 36 Nazioni differenti. Come si può notare dalla figura 2.1 che segue, l'Italia è tra i paesi maggiormente presenti. La presa dati è prevista cominciare nel 2007.

CMS è un grande rivelatore realizzato per studiare le particelle (fotoni, elettroni, muoni, protoni, ecc.) generate dalle collisioni di protoni che avvengono all'interno dell'accelleratore LHC, identificandone la tipologia e misurandone una serie di parametri (energia, impulso, direzione, ecc.). Dallo studio delle particelle saranno effettuate numerose misure di processi fisici che hanno lo scopo principale di scoprire nuovi fenomeni fisici. Tra i più importanti argomenti di studio, la ricerca del Bosone di Higgs [11] e di particelle supersimmetriche [12].

L'accelleratore LHC permetterà di ricreare condizioni simili a quelle dell'universo appena $10^{-12}s$ dopo il Bing-Bang che corrisponde ad una temperatura dell'ordine di 10^{16} gradi. I fasci di protoni saranno iniettati in LHC sfruttando l'azione combinata di un acceleratore lineare capace di accelerare

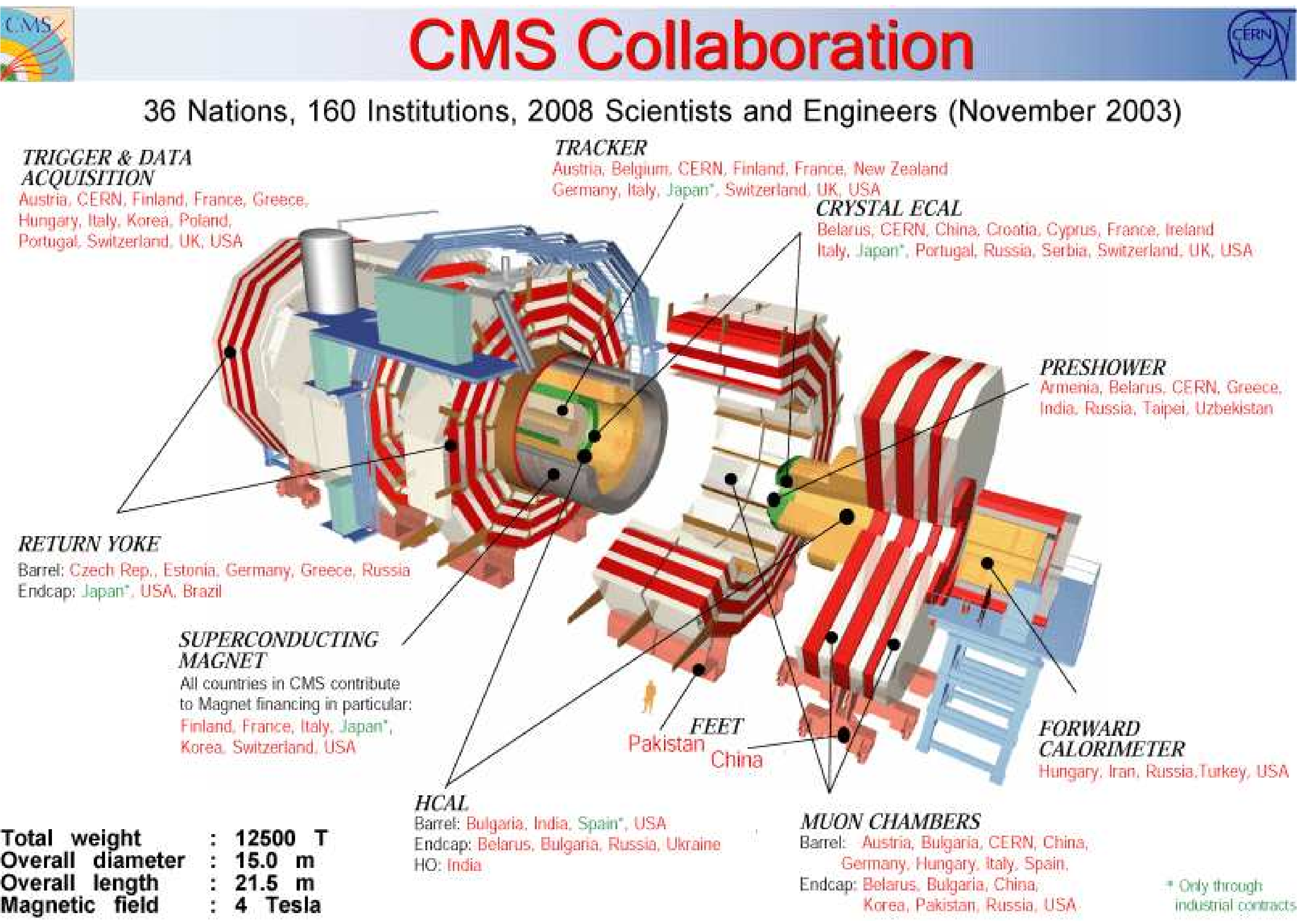

Figura 2.1:

i protoni fino a 50 MeV([1]), un ProtoSincrotone (PS) fino a 26 GeV([2]), un SuperProtoSincrotone (SPS) fino a 450 GeV, per poi essere iniettati in LHC fino a raggiungere un'energia di circa 14TeV([3]).

I fasci di protoni si incontreranno, lungo l'anello di accelerazione, in quattro regioni di intersezione dove saranno collocati quattro esperimenti (Figura 2.2): ALICE (A Large Ion Collider Experiment) [4] che si occuperà delle collisioni tra ioni pesanti, LHCB (Large Hadron Collider beauty experiment) [5] che studierà la fisica di mesoni B, CMS (Compact Muon Solenoid) ed ATLAS (A Toroidal LHC ApparatuS) [20] che hanno scopi più generali.

Ad LHC è prevista una prima fase di lavoro ad una luminosità[4] istantanea di 2×10^{33} $cm^{-2}s^{-1}$ (fase di bassa luminosità) che sarà gradualmente aumentata nel corso degli anni fino ad un massimo di 10^{34} $cm^{-2}s^{-1}$ (fase di alta luminosità). Per raggiungere questi alti valori di luminosità ogni fascio di protoni sarà costituito da 2835 pacchetti (*bunches*) contenenti in media circa 10^{11} particelle ciascuno. L'incrocio tra fasci (*bunch crossing*) avverrà ogni 25 ns. Di queste interazioni, ciascuna delle quali darà luogo a centinaia di particelle, solo una parte produrrà eventi interessanti; la restante costituirà l'inevitabile fondo di eventi a basso impulso trasverso, tecnicamente detti eventi di *minimum bias*, dovuti a collisioni "periferiche" tra protoni.

Alla luce di quanto detto, i rivelatori installati ad LHC dovranno avere i seguenti requisiti:

- un'alta granularità, ossia (elementi sensibili di piccole dimensioni), quindi un gran numero di canali di rivelazione per garantire una efficiente ricostruzione delle tracce e individuare con elevata precisione le coordinate del punto di passaggio di una particella;

- resistenza alle radiazioni in quanto l'alta energia del fascio combinata con l'elevata luminosità creerà un ambiente di radiazione parti-

[1] 1 Mega-eV= 10^6 eV (electron-Volt)

[2] 1 Giga-eV= 10^9 eV

[3] 1 Teta-eV= 10^{12} eV

[4] La luminosità un parametro fondamentale, oltre all'energia, per valutare le prestazioni di un anello di accumulazione ed è definita come il numero di collizioni nell'unità di tempo per l'unità di sezione d'urto.

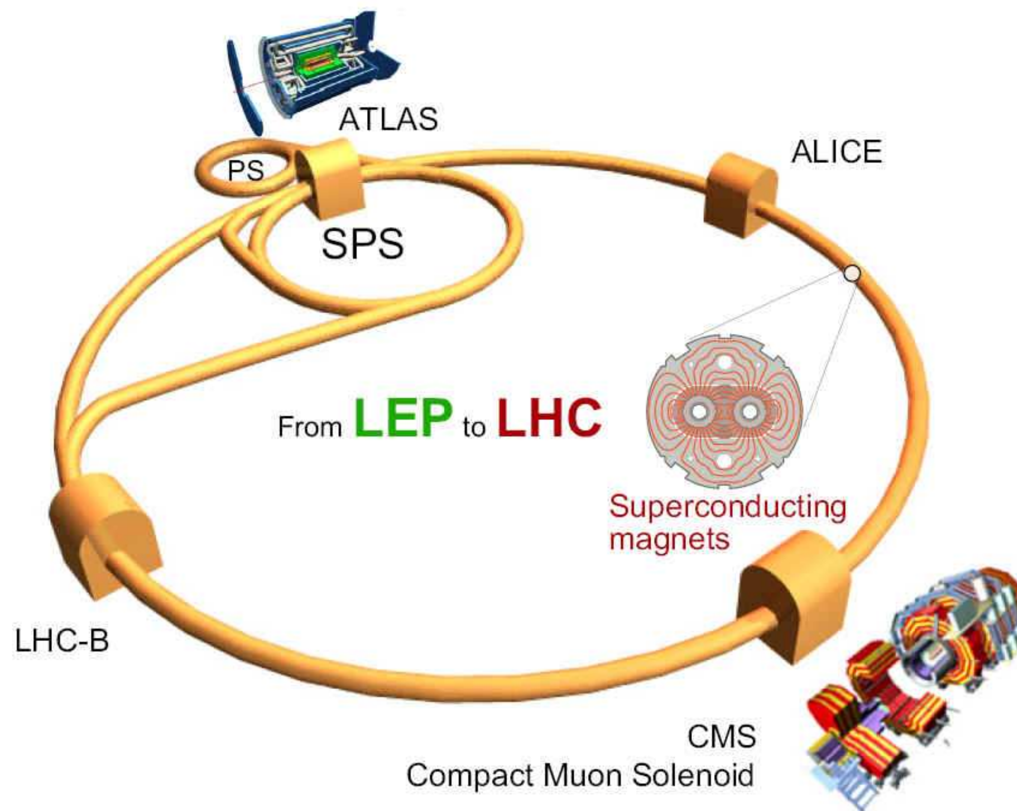

Figura 2.2: L'anello LHC ed i suoi 4 esperimenti.

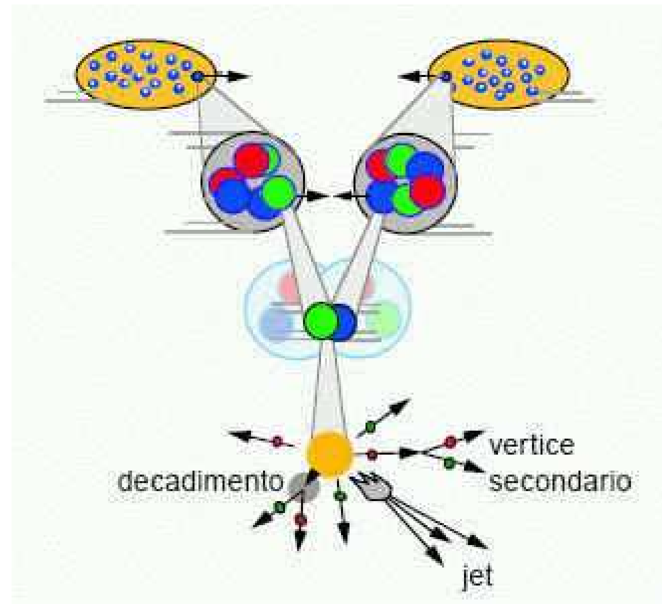

Figura 2.3: Schema di interazione tra i fasci di protoni in LHC.

colarmente elevato soprattutto per i rivelatori più vicini al punto di collisione;

- un'elettronica di lettura veloce per evitare la sovrapposizione (*pile up*) dei segnali relativi a *bunch crossing* diversi.

2.1 Il rivelatore CMS

CMS rappresenta uno dei 4 apparati sperimentali previsti ad LHC e dovrà soddisfare i requisiti sopra citati. I principali scopi fisici saranno: la ricerca del bosone di Higgs nell'intervallo di massa che va da circa 114 GeV/c^2 (limite inferiore posto dalle più recenti misure sperimentali fatte su LEPII [6]) a circa 1 TeV/c^2 (limite superiore teorico). Il Bosone di Higgs rappresenta un ingrediente fondamentale del *"Modello Standard di unificazione elettro-*

debole[5]" [19] che ne prevede l'esistenza. Tuttavia, non esiste ancora evidenza sperimentale della sua esistenza. CMS condurrà anche la ricerca di particelle supersimmetriche [12], lo studio della fisica dei mesoni B e lo studio dei processi di produzione del decadimento del quark *top*. Questi processi saranno studiati attraverso una accurata identificazione di muoni, elettroni, fotoni su un ampio intervallo dell'impulso.

2.2 Struttura dell'apparato sperimentale

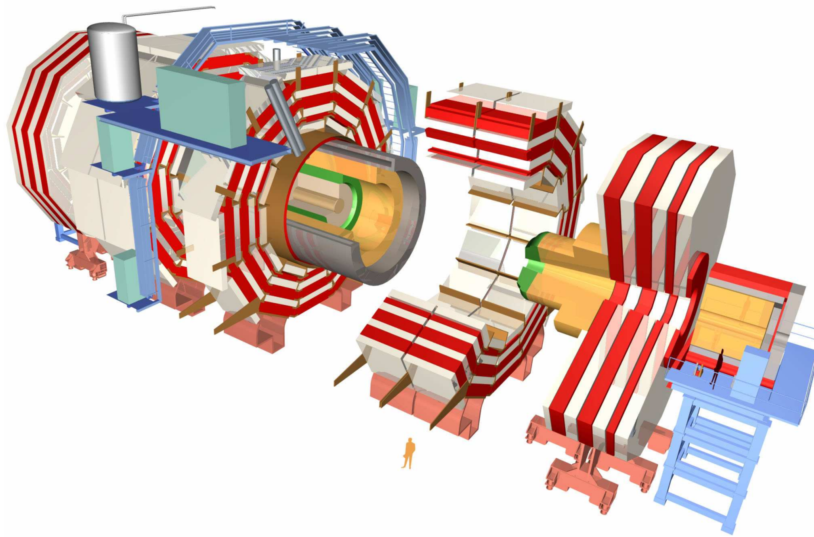

Figura 2.4: Vista tridimesionale del rivelatori dell'apparato CMS.

L'apparato CMS (Figura 1.2), lungo circa 22 m, con diametro di 14.6 m e del peso di 14500 tonnellate, si sviluppa secondo una simmetria cilindrica intorno al punto di intersezione dei fasci ed è costituito da diversi rivelatori alloggiati l'uno all'interno dell'altro. Il sistema cartesiano di riferimento convenzionalmente adottato è il seguente: l'asse x è orizzontale e punta dal

[5]Secondo questa teoria, tutte le particelle devono la loro massa all'interazione con il campo di Higgs.

verso il centro di LHC, l’asse y è perpendicolare all’asse x e alla direzione del fascio e punta verso l’alto, l’asse z infine è allineato secondo la direzione del fascio e del campo magnetico di CMS.

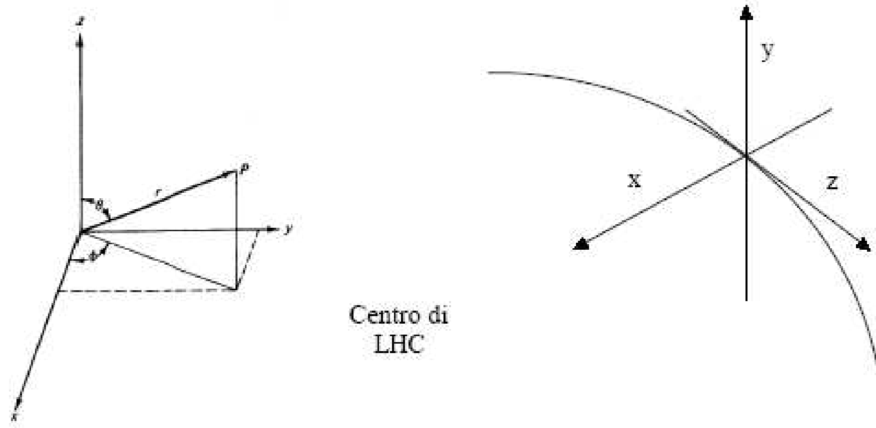

Figura 2.5: sistema di riferimento cartesiano e polare per CMS.

La parte del rivelatore parallela all’asse del fascio, lungo l’asse z, è detta “*barrel*” ed è costituita da superfici cilindriche, mentre quelle perpendicolari al fascio, “*endcap*” (tappi), sono costituite da superfici piane a forma di disco.

Partendo dall’interno verso l’esterno si distinguono:

- un sistema di tracciamento (*Tracker*) ad alta precisione che circonda il tubo a vuoto del fascio (*beam pipe*) intorno alla zona di interazione.
- un calorimetro elettromagnetico (E-CAL);
- un calorimetro andronico (H-CAL);
- un magnete solenoidale superconduttore;
- un sistema di rivelazione dei muoni;

2.3 I sottorivelatori di CMS

2.3.1 Il Tracker

È il sistema di rivelazione più vicino al punto di interazione. I principali scopi fisici sono:

- la ricostruzione delle tracce di particelle cariche;
- la ricostruzione dei vertici primari e secondari di decadimento;
- l'identificazione di muoni ed elettroni insieme al sistema di rivelatori di muoni e ai calorimetri;

La complessità delle interazioni prodotte ad LHC, le condizioni in cui CMS si troverà a lavorare impongono prestazioni eccellenti al sistema di tracciamento. Per ottenere una buona ricostruzione delle traccie e una buona risoluzione delle misure del momento trasverso è richiesto ai rivelatori un gran numero di canali di lettura. La quantità di materiale attraversato dalle particelle dovrà essere la più bassa possibile per evitare interazioni secondarie e conversioni che altererebbero le prestazioni del rivelatore. Infine, tutti i rivelatori del Tracker devono avere una buona risoluzione temporale al fine di distinguere gli eventi che provengono da *bunch crossing* consecutivi. Per soddisfare queste richieste il Tracker è costruito con rivelatori al silicio che assicurano le prestazioni richieste in termini di risoluzione spaziale (fino a pochi μm), rapidità di risposta (dell'ordine di 10 ns) e tolleranza alle alte dosi di radiazione.

Il Tracker ha un raggio massimo di 110 cm e lunghezza totale di 540 cm. In esso si distinguono una regione centrale coassiale al fascio (*barrel*) costituita da 13 strati cilindrici (*layer*), e due regioni terminali laterali piane perpendicolari al fascio, di cui una interna al *barrel* (*mini-endcap*) ed una esterna (*endcap*), costituite complessivamente da 28 dischi, 14 per lato.

I rivelatori del *barrel* sono rettangolari con le *strip* parallele alla direzione del fascio, mentre i rivelatori degli *endcap* sono trapezoidali con le *strip* radiali. In questa maniera i rivelatori del *barrel* forniscono le misure delle coordinate ϕ ed r dei punti di impatto delle tracce mentre i rivelatori degli

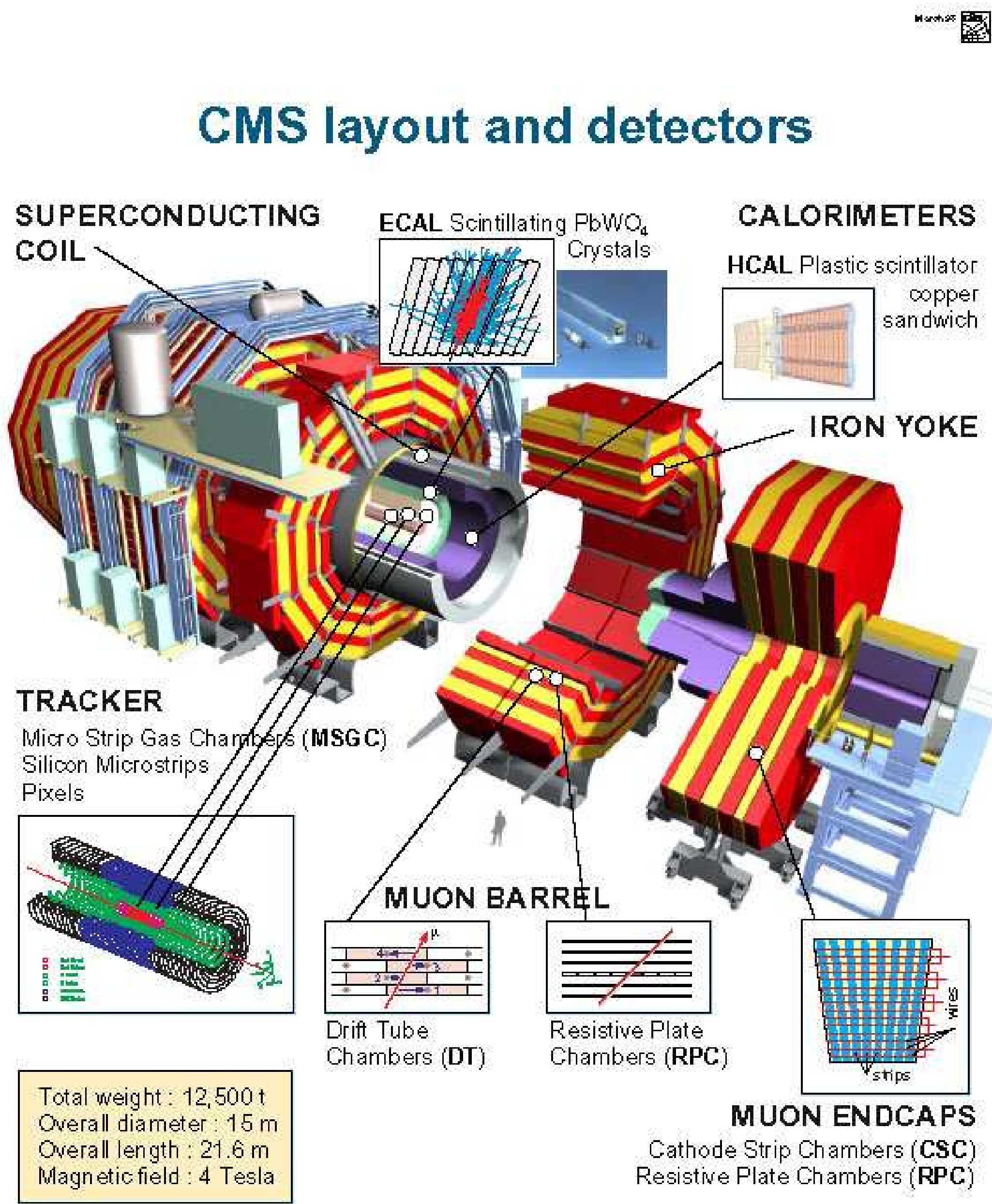

Figura 2.6: I rivelatori dell'apparato CMS.

endcap forniscono le informazioni su ϕ e z. La parte più interna adotta rivelatori a pixel di silicio, Silicon Pixel Tracker (SPT) che consiste di tre strati nella regione del *barrel* e due dischi nella regione *endcap*, come mostrato in figura 2.7.

La regione esterna del Tracker utilizza rivelatori a microstrisce al silicio, detti *Silicon Strip Tracker* (SST). Qui la densità delle tracce diventa più bassa mentre aumenta con la distanza dalla *beam pipe* la superficie sensibile. Una minore granularità è sufficiente per far fronte alle prestazioni richieste e a limitare il numero dei canali di lettura.

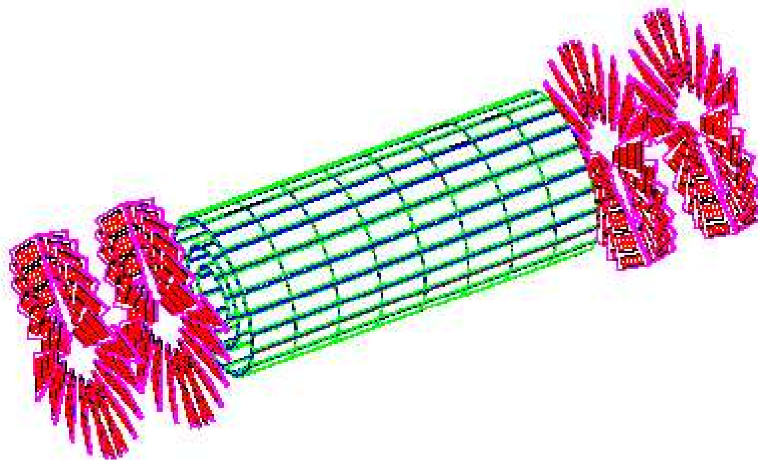

Figura 2.7: Il rivelatore a pixel di silicio.

2.3.2 Il sistema calorimetrico

Il sistema calorimetrico identifica elettroni, fotoni e jets[6] adronici (vedi sez. 3.6), misurandone energia e posizione. Esso ha un tempo di risposta infe-

[6] Il nome che i fisici danno ad un gruppo di particelle che emergono da una collisione o decadimento, tutte viaggianti all'incirca nella stessa direzione e che trasportano una significante frazione di tutta la energia nell'evento. Le particelle nei jet sono principalmente particelle composte da costituenti soggetti alla "interazione forte", chimate adroni.

riore a 50 μs e possiede un'elevata granularità, allo scopo di minimizzare la sovrapposizione delle tracce (*pile-up*) che ne ridurrebbe l'efficienza totale.

Il sistema calorimetrico è costituito da un calorimetro elettromagnetico (ECAL) ed uno adronico (HCAL) che assorbono completamente le particelle che interagiscono in essi e ne misurano l'energia. Il calorimetro elettromagnetico è costituito da cristalli di tungstanato di piombo ($PbWO_4$) che offrono le migliori prestazioni per l'identificazione e la misura accurata dell'energia di fotoni ed elettroni anche in condizioni di intenso campo magnetico ed elevata dose di radiazioni, e con tempi di risposta brevi (circa 10 ns). Il calorimetro elettromagnetico riveste un importanza particolare per la ricerca del bosone di Higgs, in particolare nel caso in cui si considera il decadimento del bosone di Higgs in due fotoni o quattro elettroni.
I calorimetri adronici sono invece realizzati alternando strati di materiale assorbitore in ottone a strati di scintillatori (4 mm).

2.3.3 Il magnete

Il sistema calorimetrico insieme a quello tracciante è alloggiato in un magnete costituito da un solenoide superconduttore lungo 13 m con un diametro interno di 2.95 m che sarà capace di produrre un campo magnetico di 4 T, parallelo alla direzione dei fasci di protoni collidenti e che spingerà le particelle cariche, risultanti dalle collisioni protone–protone su traiettorie elicoidali tali da consentire misure precise del loro momento nel sistema di tracciamento.

2.3.4 Il sistema di rivelazione dei muoni

La parte esterna di CMS è costituita dal rivelatore di muoni, composto da quattro "stazioni" centrali e otto (quattro per parte) laterali, che costituiscono il *barrel* e gli *endcap* rispettivamente.

Il rivelatore di muoni è posto esternamente ai calorimetri e alla bobina del solenoide, e consiste di stazioni inserite tra gli anelli di ritorno del magnete. Il momento dei muoni può essere misurato in tre modi diversi: attraverso la curvatura della traiettoria nel tracker; attraverso l'angolo di curvatura all'esterno della bobina; attraverso la traiettoria negli anelli di ritorno. Le ultime

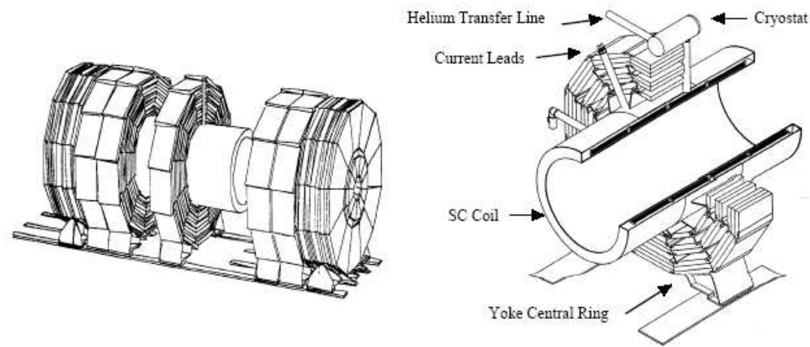

Figura 2.8: Vista del magnete e particolare dell'anello centrale con la bobina del solenoide.

due determinazioni sono completamente indipendenti dalla misura effettuata nel tracker, la loro combinazione consente una buona risoluzione e rende il sistema, data la ridondanza, estremamente robusto in presenza di fondo anche a luminosità elevate. La risoluzione spaziale richiesta ad ogni stazione è di almeno 100 μm. I rivelatori utilizzati all'interno delle singole stazioni sono diversi a seconda che queste siano nel *barrel* o negli *endcaps*: la bassa frequenza (10 Hz/cm^2) e il basso campo magnetico rendono possibile l'utilizzo di tubi a deriva (DT) nel *barrel*; al contrario negli *endcaps* l'alta frequenza e l'intenso, non uniforme, campo magnetico indicano l'utilizzo di rivelatori CSC (Cathode Strip Chamber). Ogni stazione è inoltre equipaggiata con RPC (Resistive Plate Chamber) che hanno eccellente risoluzione temporale associata ad una discreta risoluzione spaziale per fornire un rivelatore di trigger (vedi sez. 2.3.5 di muoni dedicato, necessario, data l'alta frequenza e fondo attesi per LHC.

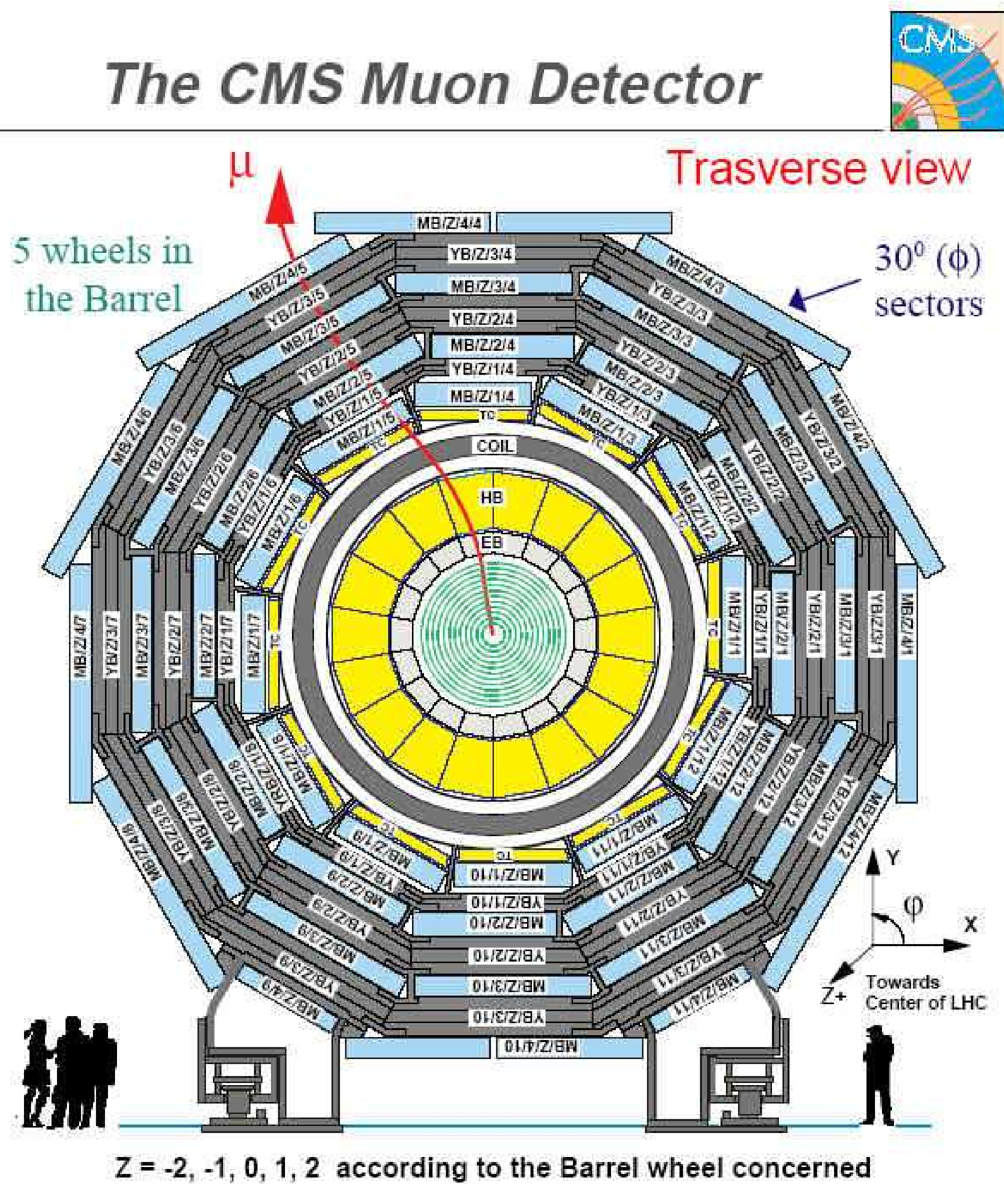

Figura 2.9: Vista Trasversale del rivelatore di muoni CMS.

2.3.5 Il sistema trigger

L'incrocio dei fasci di LHC produce circa 40 milioni di eventi al secondo[7] di cui solo una piccola parte darà luogo ad eventi interessanti per gli studi di Fisica. Le tecnologie attualmente disponibili per la memorizzazione permanente su memorie magnetiche non possono fornire una velocità di scrittura e spazi disco sufficienti per riuscire a memorizzare tutte le informazioni provenienti dal rivelatore; inoltre, pur ipotizzando di avere strumenti tecnologici adatti, memorizzare tutti gli eventi comporterebbe soltanto spreco di risororse, tempo e danaro. Per tutti questi motivi risulta indispensabile un sistema di presa dati in grado di effettuare in tempo reale (*on-line*) una selezione tra eventi d'interesse che necessitano di essere memorizzati sul disco ed eventi di fondo di scarso interesse. Questo sistema in grado di effetuare una selezione (parziale) degli eventi prende il nome di "*trigger*". Il sistema di trigger dell'esperimento CMS utilizza una architettura a due livelli. Il primo livello di trigger (L1A) è progettato in modo da produrre al massimo 10^5 eventi al secondo (100 KHz) di tasso di uscita. L'elevata frequenza di *bunch crossing* rende impossibile prendere una decisione tra un *bunch-crossing* e quello successivo. Per questo motivo, i sensori devono essere in grado di memorizzare temporaneamente i dati in una coda (*pipeline*) in attesa delle decisioni provenienti dal trigger L1A. Il tempo di attesa prende il nome di latenza e il suo valore massimo è stato fissato in 3.2 μs., pari a 120 *bunch-crossing*. La elaborazione dei dati operata dal trigger di primo livello è completamente hardware, ovvero gli algoritmi impiegati per la riduzione del flusso di dati sono realizzati mediante reti logiche integrate sui componenti del sistema. Il successivo livello di trigger di alto livello (HLT) riduce il tasso di 100 KHz di eventi generato da L1A, al tasso finale di 100 Hz. Il sistema di lettura (*read-out*) può memorizzare gli eventi in uscita dal L1A, mentre il trigger di secondo livello (HLT) decide se memorizzae o rigettare gli eventi. Una caratteristica dell'esperimento CMS è la realizzazione del trigger di alto livello senza reti logiche dedicate allo scopo, ovvero esclusivamente tramite calco-

[7]in realtà per ogni collisione tra due pacchetti di protoni si hanno circa 20 interazioni, ma il sistema di trigger non è in grado di identificarli e separarli. Pertanto, ai fini della scrittura dei dati su disco, le 20 interazioni sono viste come un unico "evento".

latori programmati con opportuno software. Gli algoritmi impiegati da HLT per ottenere questa drastica reiezione devono essere in grado di processare gli eventi in un tempo dell'ordine di 10^{-2}s, e saranno installati in una *farm* di calcolatori con migliaia di nodi. Per fornire la connettività tra il sistema di *read-out* e i processori della *farm*, è utilizzata una rete a commutazione. Tale rete deve essere un grado di sostenere un flusso di dati di circa 1 Tb/s[8].

[8] 1 Tera–bit=$10^1$2bit

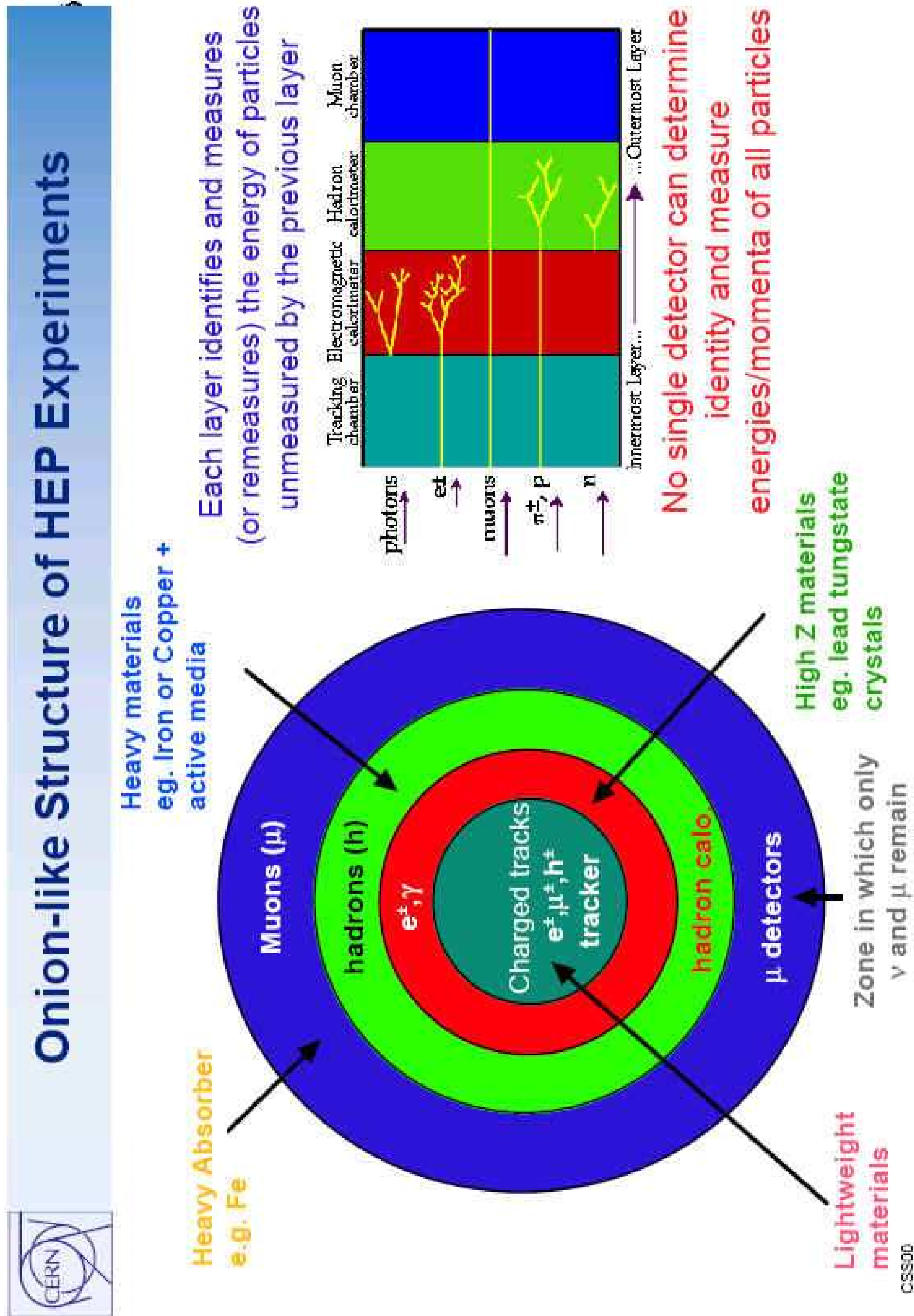

Figura 2.10:

Figura 2.11: Fotografia dell'apparato CMS in costruzione.

Figura 2.12: Fotografia dell'apparato CMS in costruzione.

Capitolo 3

Il software di CMS

Nei più recenti esperimenti di fisica delle alte energie strumenti di ingegneria del software e linguaggi orientati ad oggetti sono diventati di uso comune per lo sviluppo di applicazioni quali programmi di simulazione e ricostruzione degli eventi e dell'analisi fisica finale.

La quantità di dati che ogni esperimento produrrà all'anno è dell'ordine del Pbyte[1], un ordine di grandezza mai raggiunto prima dai precedenti esperimenti. I sistemi software su larga scala come quello di CMS richiedono un'architettura ben definita ed omogenea; applicazioni ripartite in entità gestibili indipendentemente (componenti/moduli), interfacce e protocolli definiti in modo univoco allo scopo di implementare una efficiente comunicazione tra le parti. Il linguaggio di programmazione adottato in CMS è il C++.

3.1 Le componenti del software CMS

Diverse componenti software sono necessarie nell'esperimento per le diverse fasi di operazione, che vanno dall'acquisizione dei dati prodotti in seguito alle collisioni, fino all'analisi dei dati che costituisce l'ultimo passo per le misure di Fisica e le possibili nuove scoperte. Il funzionamento dell'apparato può essere schematizzato logicamente come segue:

[1] 1 Peta–Byte $\approx 10^{15}$ Byte

1. *I fasci di protoni vengono fatti collidere:* a seguito delle collisioni vengono prodotte particelle secondarie che interagiranno con i sottorivelatori costituenti l'apparato CMS. Le particelle cariche saranno rivelate dal Tracker; fotoni ed elettroni prodotti dalle collisioni depositeranno la maggior parte della loro energia nel calorimetro elettromagnetico (E-CAL); la paricelle adroniche, invece, rilasceranno solo una parte della loro energia nel calorimetro elettromagnetico e depositeranno una quantità maggiore di energia nel calorimetro adronico; le uniche particelle in grado di attraversare il Tracker, i calorimetri e raggiungere il sottorivelatore più esterno sono i muoni (figura 3.1).

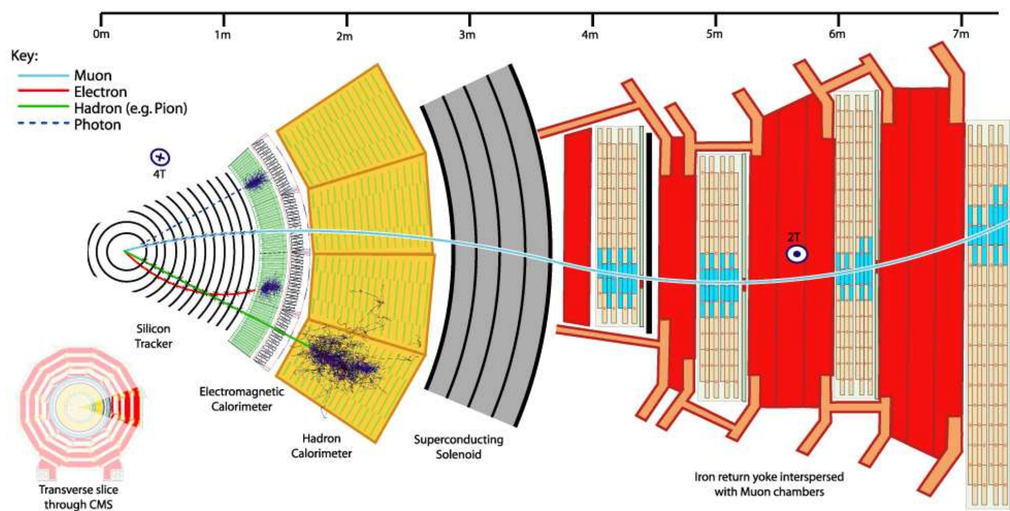

Figura 3.1: Una sezione della sezione trasversale del rivelatore: sono schematizzate particelle come fotoni ed elettroni, che si fermano nel calorimetro-elettromagnetico, gli adroni (come ad esempio i protoni) che raggiungono nel calorimetro adronico, mentre i muoni risultano le uniche particelle in grado di attraversare i primi sottorivelatori giungendo fino alle apposite "camere dei muoni".

2. *Acquisizione dati:* tutte le informazioni provenienti dai sottorivelatori che costituiscono l'apparato vengono raccolte dall'elettronica di lettura, e scritte sul disco, se il sistema trigger da un segnale positivo. Il trigger

(vedi sez. 2.3.5) effettua uno scarto di molti degli eventi che non risulteranno utili ai fini dell'analisi fisica; tali inforamzioni sono tecnicamente indentificate con il nome di "RawData". È utile quì ricordare che la componente HLT del trigger è costituita esclusivamente da software.

3. *Ricostruzione:* vengono "ricostruite" le particelle dello stato finale sulle quali si basa l'analisi fisica a partire dai i "RawData" (memorizzati sul disco in fase di acquisizione) ed adoperando appositi algoritmi di ricostruzione che partono dai segnali elementari di ciascun rivelatore.

4. *Analisi Fisica:* basandosi sul modello ad oggetti dei dati creato in fase di ricostruzione e sulle conoscenze attuali della fisica delle alte energie si effettuano misure di processi fisici e si cercano evidenze di nuove particelle (es: Bosone di Higgs, particelle Supersimmetriche, etc.).

3.2 L'architettura software di CMS

Il disegno generale della Architettura Software di CMS è motivato dai seguenti principi:

- **ambienti multipli:** gli stessi moduli software devono essere in grado di girare in una varietà di ambienti: nella fase *on-line*, ossia la fase di acquisizione dati e di trigger, durante la fase *off-line*, ovvero durante la ricostruzione, ed in fase di analisi finale.

- **sviluppo distribuito del codice:** il software viene sviluppato in maniera distribuita da gruppi di persone sparse geograficamente;

- **flessibilità:** il software deve essere flessibile, ossia facilmente adattabile, poichè non tutti i requisiti possono essere noti in anticipo;

- **semplicità d'uso:** il software deve essere facilmente usabile dalla collaborazione dei fisici senza imporgli di diventare degli esperti di informatica.

I requisiti della architettura del software portano alla seguente struttura per il software di CMS:

- **un framework:** che definisce le astrazioni di livello più alto, il loro comportamento e i modelli (*patterns*) di collaborazione;
- **moduli software di Fisica:** scritti da gruppi fisici che possono essere inseriti nel *framework*, anche a *run-time*.

3.2.1 I Framework in generale

Un *framework* è un insieme di classi cooperanti riusabili che forniscono lo scheletro di un'applicazione per uno specifico dominio applicativo. Il *framework* delinea l'architettura delle applicazioni in cui viene usato. Definisce la struttura generale, le classi e gli oggetti che ne fanno parte, le loro responsabilità principali, il modo in cui classi e oggetti collaborano, e il flusso di controllo. Un *framework* che affronti le scelte progettuali adopernado efficientemente i *design patterns* ha maggiori probabilità di ottenere alti livelli di riuso del codice. Un'ulteriore vantaggio che deriva dall'utilizzo dei *design patterns* è la migliore documentabilità che rende l'apprendimento più semplice da parte degli sviluppatori, in particolare per coloro che già conoscono la filosofia dei *pattern* adoperati all'interno del framework stesso [22].

3.2.2 Il Framework di CMS: COBRA

Il *framework* di CMS è denominato **COBRA** (Coherent ObjectOriented Base for Simulation RecCollection and Analysis). COBRA rappresenta lo scheletro dell'architettura software, e su di esso poggiano i due pacchetti OSCAR e ORCA che gestiscono rispettivamente le fasi di simulazione e ricostruzione dell'esperimento, che sono descritte descritte nei paragrafi 3.4 e 3.5.2. Il *framework* di CMS si occupa di gestire il flusso dei dati durante queste fasi garantendo contemporaneamente un'interfaccia uniforme su tutto il codice di CMS.

COBRA è in grado di gestire le sessioni tipiche di un software di fisica alle alte energie: strategie di ricostruzione, accesso ai dati degli eventi di collisione, gestione della persistenza, accesso ai dati asincroni (calibrazione), definizioni delle impostazioni dei rivelatori (costanti geometriche, etc.), interfacce utente.

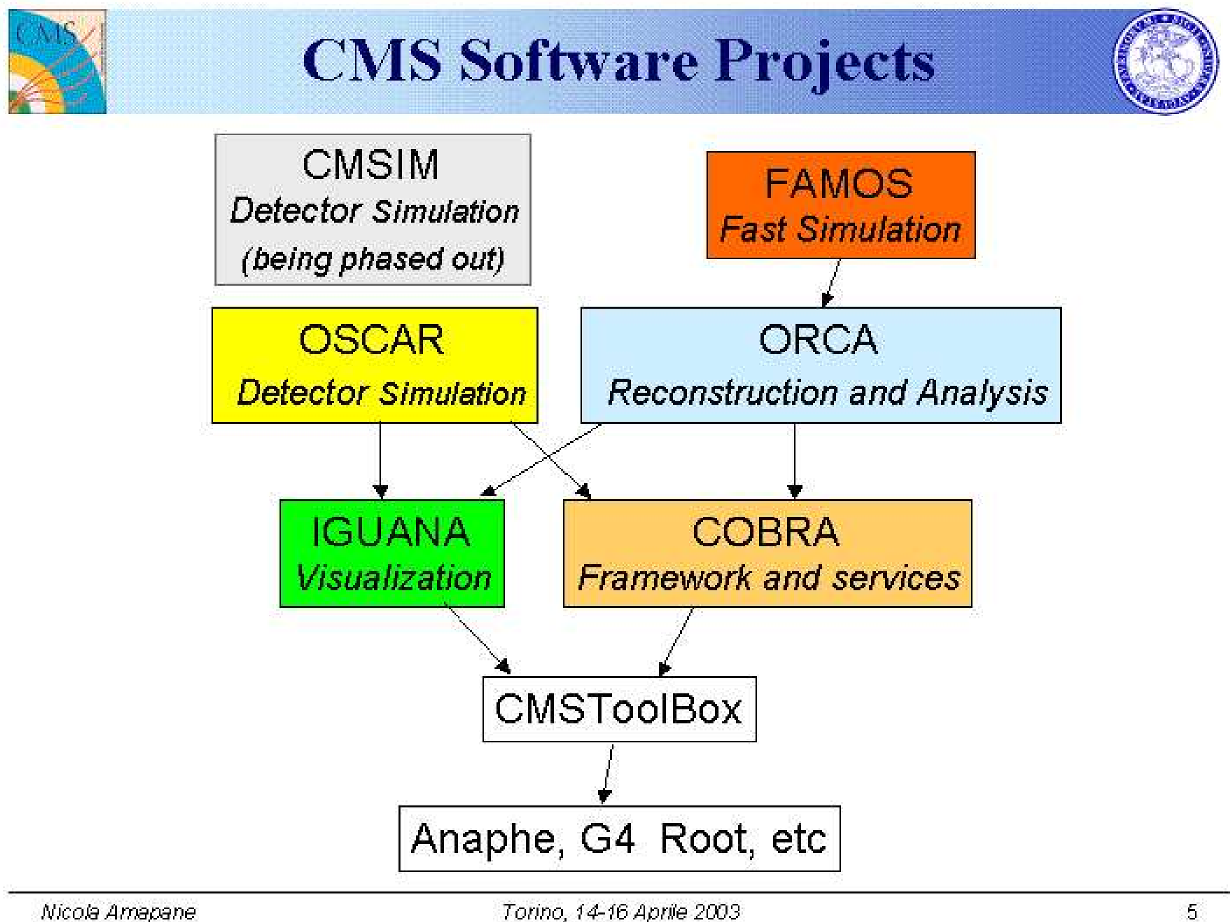

Figura 3.2: I pacchetti software coinvolti nell'esperimento CMS.

3.3 Schema delle attività principali dell'esperimento.

Le attuali attività principali per il software di CMS sono quindi:

1. **Simulazione:**
   - simulazione degli eventi; (PYTHIA [17] ed altri generatori)
   - simulazione del rivelatore; (OSCAR)
   - simulazione dell'elettronica o digitizzazione (ORCA)
2. **Calibrazione e Ricostruzione:** (ORCA)
   - Ricostruzione nei sottorivelatori: Tracker, E-CAL, H-CAL, Camera per muoni;
   - Ricostruzione combinata tra diversi rivelatori;
   - Algoritmi di decisione sull'acquisizione degli eventi (HLT)
   - Sono previsti particolari applicazioni della ricostruzione che consistono nel calcolo di costanti caratteristiche dei rivelatori (ad esempio allineamenti, fattori di scala nelle misure di energia, etc.) che permettono di ottimizzare le prestazioni dei rivelatori.
3. **Analisi:**
   - Ricerca di Higgs;
   - Studio dei mesoni B;
   - Studio della superssimetria;
     ...ricerca di altre tipologie di particelle e studio di nuove teorie.

3.4 La simulazione

Attualmente l'acceleratore LHC e lo stesso rivelatore CMS sono ancora in fase di istallazione, per cui i dati dell'esperimento non sono ancora disponibili[2];

[2]Probabilmente lo sarranno a partire dal 2007.

pertanto in questi anni si sta cercando di simulare tutti i possibili canali di fisica di interazione che potranno presentarsi. In tal modo è possibile lavorare alle fasi successive (ricostruzione ed analisi) anche in assenza dei dati reali in maniera tale che quando i dati saranno disponibili, gli strumenti software necessari saranno pronti. La simulazione rappresenta uno degli aspetti fondamentali dell'esperimento anche durante la presa dati perchè permette di studiare diversi canali di fisica separatamente, in modo da poterli identificare quando si presentano mescolati insieme nei campioni di dati reali. Il processo di simulazione comprende tre momenti dell'esperimento:

- Simulazione dei processi di collisione e generazione delle particelle dello stato finale;
- Simulazione dell'interazione con la materia che costituisce i rivelatori delle particelle dello stato finale;
- Simulazione della risposta dei rivelatore (digitizzazione) e dell'elettronica di acquisizione.

Attualmente il pacchetto adoperato per simulazione della fisica delle particelle è CMKIN basato su PYTHIA [17]. L'interazione delle particelle con la materia viene simulata attraverso OSCAR (Object Oriented Simulation for CMS Analysis and Reconstruction), un pacchetto Object-Oriented basato sul toolkit GEANT4 [21] sviluppato al CERN. OSCAR legge gli eventi forniti da CMKIN e simula gli effetti di perdita di energia (bremsstrahlung, ionizzazione, ...), diffusione multipla, sciami elettromagnetici o adronici nei materiali del rivelatore. L'informazione è immagazzinata in forma di oggetti chiamati *hit*[3] e contiene tutti i dettagli necessari per simulare la risposta del rivelatore.

Per i rivelatori traccianti (parte interna del Tracker, e il sistema muonico) l'informazione immagazzinata si riferisce ai punti di ingresso e di uscita della traccia, al vettore momento delle particelle nel punto d'ingresso, al tipo di particella, all'energia persa nel volume sensibile.

Per i calorimetri gli *hit* simulati contengono l'informazione relativa all'energia depositata in un cristallo o nello scintillatore.

[3]Un hit rappresenta il segnale identificato su di una cella del rivelatore

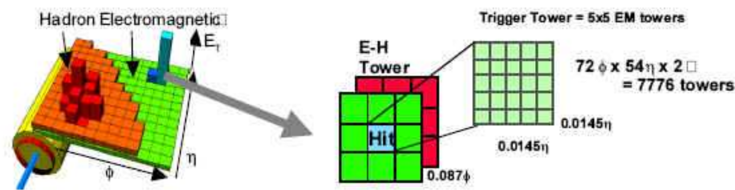

Figura 3.3: Grafica dei rivelamenti effettuati nei calorimetri H-CAL (rosso) E-CAL (verde).

I risultati della simulazione vengono successivamente importati nel software di ricostruzione, "ORCA"[4] per la fase di digitizzazione.

Per simulare la risposta del rivelatore è necessario combinare il segnale relativo all'evento di interesse con il segnale prodotto dagli eventi di fondo (eventi di *pile-up*) che risultano essere in media una ventina per ogni *bunch crossing*. Oltre ai 20 eventi contemporanei a quello di interesse vengono sovrapposte le risposte del rivelatore ad eventi scelti in maniera casuale che sono avvenuti nei 5 *bunch crossing* precedenti e nei 3 successivi rispetto all'evento di interesse. Ad ogni *hit* viene assegnato il tempo del *bunch crossing* a cui appartiene. Il numero medio di eventi di *pile-up* sarà quindi pari a 160, 20 per ciascuno degli 8 pacchetti vicini a quello contenente il segnale di interesse. Gli eventi di *pile-up* sono presi a caso da una collezione di eventi simulati di *miminum bias* il cui numero è scelto abbastanza grande per assicurare che nello stesso evento di interesse non venga utilizzato per due volte lo stesso evento che invece può essere usato in eventi diversi. Questo metodo di riusare gli stessi eventi di *pile-up* è stato scelto per limitare le necessità di tempo di CPU necesaria per simulare eventi di fondo.

La fase, in cui si fa una accurata simulazione della risposta del rivelatore, è detta **fase di digitizzazione**; in particolare si tiene conto del modo in cui viene acquisita l'informazione da parte dell'elettronica (ad esempio possono essere aggiunti gli effetti dovuti al rumore). Nella fase di digitizzazione gli *hits* simulati da OSCAR, chiamati *"SimHits"* vengono codificati per ottenere

[4]Object Oriented Reconstruction for CMS Analysis

i cosidetti *"DIGI"* o *"RawData"*, che sono identici, in quanto a tipologia di informazione, a quelli che saranno raccolti quando l'acceleratore entrerà in funzione. La dimensione finale di un singolo evento è in media di 2 MB.

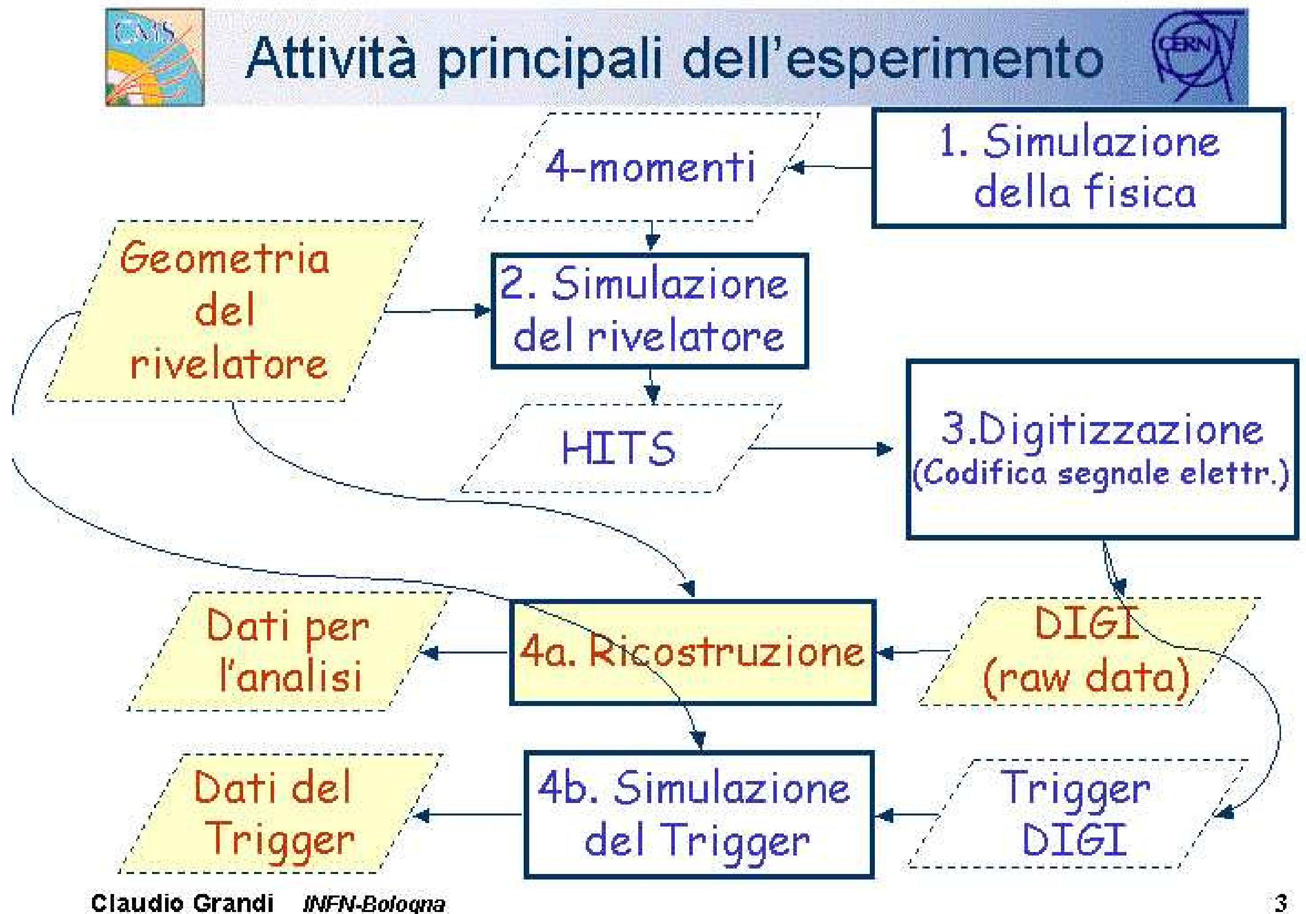

Figura 3.4: Diagramma di flusso delle fasi dell'esperimento.

3.5 Il processo di Ricostruzione

La fase di ricostruzione provvede alla creazione, a partire dalle informazioni fornite dai sottorivelatori, di oggetti che sono utilizzabili per l'analisi fisica finale.

Per ogni collezione di oggetti che si tenta di ricostruire esiste spesso più di un algoritmo; ciò consente di valutare e confrontare diverse strategie di ricostruzione. La ricostruzione degli oggetti dell'analisi può avvenire in due modi distinti: sia basandosi sulle informazioni provenienti da un'unico sottorivelatore, sia combinando le informazioni provenienti da più sottorivelatori. All'interno del Tracker è possibile individuare tracce e vertici (primari e secondari) delle particelle: i depositi di energia lasciati dalle particelle nei calorimetri elettromagnetico e adronico sono raggruppati in base a criteri di adiacenza, per formare gruppi di depositi detti "*cluster*". A partire dai *cluster* è possibile ricostruire rispettivamente elettroni e fotoni nel primo calorimetro (E-CAL), e sciami adroni nel secondo (H-CAL); attraverso il sottorivelatore più esterno costituito dalle camere per muoni è possibile identificare esclusivamente muoni.

Al tempo stesso, però, le ricostruzioni basate sui singoli rivelatore vanno integrate per poter risalire alle informazioni complete di tutte le particelle. Ad esempio le informazioni provenienti dalle interazioni di elettroni all'interno del calorimetro elettromagnetico vanno integrate con le misure del momento effettuate nel Tracker; analogamente, per poter risalire ad un quark o gluone si possono adoperare i *jet* (vedi sez. 3.6), che a loro volta possono essere ricostruiti a partire dai *cluster calorimetrici e dalle tracce del Tracker*. La possibilità di una ricostruzione combinata equivale, dal punto di vista del programma, alla necessità di dover eseguire una catena di moduli di ricostruzione da eseguire consecutivamente. Le sequenze di moduli della ricostruzione possono diventare estremamente complesse in fase di presa dati. Un possibile approccio per affrontare questo problema è specificare la sequenza di moduli di ricostruzione da eseguire esplicitamente; tuttavia specificare esplicitamente la successione di operazioni da eseguire implica una serie di problematiche:

- la catena della sequenza può diventare molto complessa, e difficilmente gestibile;

- è possibile che venga eseguito del software non necessario all'analisi richiesta con conseguente rallentamento dell'esecuzione;
- è possibile che la sequenza fornita sia inesatta, e il programma riporti degli errori;
- l'ordine di esecuzione può non essere noto a priori, ma dipendere dalle richieste che vengono fatte a *run-time*.

Queste considerazioni hanno portato alla scelta di un metodo ricostruzione diverso, detto ricostruzione *"on demand"*. Questa tecnica si basa sull'invocazione (implicita) degli oggetti necessari alla ricostruzione, ossia ogni modulo che può prendere parte alla ricostruzione verrà adoperato solo se effettivamente richiesto (da qui il nome "ricostruzione su richiesta").

3.5.1 La ricostruzione *"on demand"*

A differenza diun modello in cui la sequenza delle operazioni è prestabilita, nello sviluppo dell'intero programma di CMS non esiste un ordinamento centrale delle azioni e non esiste un controllo esplicito del flusso dei dati. Esistono solo dipendenze implicite. I moduli si registrano al framework all'inizio del programma e non sono attivi fino a quando i loro servizi non sono effettivamente richiesti.

In questo modo soltanto le componenti software di cui realmente si ha bisogno verranno eseguite; il meccanismo di invocazione implicita viene innescato quando viene fornito un nuovo evento, che provoca il cambiamento dello stato interno di uno o più oggetti del programma. Ogni modulo che vuole essere "avvertito" quando avviene il cambiamento dello stato interno di uno o più oggetti del programma deve registrarsi come osservatore (***"observer"*** dello stesso presso un particolare oggetto chiamato ***"dispatcher"***. La registrzione deve avvenire all'inizio del programma, ovvero al momento della creazione degli "oggetti osservatori". Al cambiamento dello stato interno di un oggetto, il ***dispatcher*** informa tutti gli osservatori registrati. Questo meccanismo di "registrazione" e "notifica" viene implemtato implementato con l'ausilio dell'*Observer-Pattern* [22].

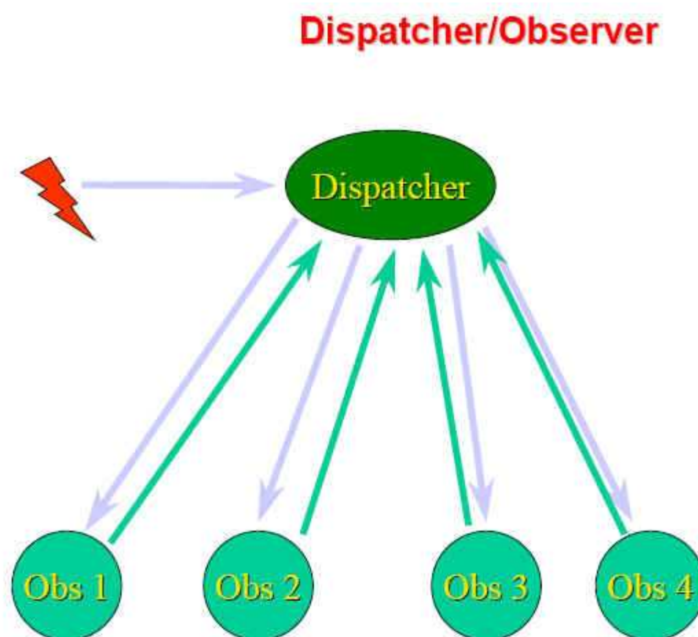

Figura 3.5: Schema dell'interazione tra Dispatcher - Observer; Le linee verdi (chiare) indicano la registrazione degli *"observer"* presso il *"dispatcher"* (fase di registrazione); Le linee viola (scure) invece, indicano l'arrivo di un nuovo evento (linea orizzontale) e la conseguente segnalazione del *"dispatcher"* a tutti gli *"observer"* registrati precedentemente (fase di notifica).

Con questo meccanismo se, ad esempio l'utente vuole effettuare l'analisi di un particolare oggetto, scrive il proprio "codice utente" che non dovrà portare conoscenza delle fasi precedenti (ricostruzione e simulazione) del processamento, ma soltanto le collezioni di oggetti che intende adoperare per l'analisi specifica. All'atto di una richiesta di alto livello viene iniziata una catena di richieste che arriva fino agli oggetti più elementari, i *RawData*; in tal modo solo gli oggetti che effetivamente debbono prendere parte alla ricostruzione dell'oggetto di analisi richiesto saranno effettivamente ricostruiti. Le richieste vengono quindi soddisfatte ripercorrendo la catena in senso inverso. I vantaggi che derivano da questa strategia sono ovviamente molteplici: innanzi tutto vi è un risparmio in termini di complessità di codice, risparmio di tempo, ma soprattutto non si corre il rischio di seguire percorsi di ricostruzione errati; inoltre, grazie al supporto per la persistenza, ossia la possibilità di registrare gli oggetti sul disco, offerto dal framework, se una collezione si trova già registrata sul disco non viene riprocessato il corrispondente modulo di ricostruzione.

3.5.2 Le fasi della ricostruzione

L'attuale pacchetto di ricostruzione di CMS è **ORCA** (Object Oriented Reconstruction for CMS Analysis), ed è usato sia per la ricostruzione degli oggetti provenienti dal rivelatore (tracce, cluster, jet, vertici, ...) che per la ricostruzione degli oggetti fisici (elettroni, muoni,..). Il processo di ricostruzione è visto come una collezione di unità di ricostruzione indipendenti (***RecUnit***), ognuna delle quali produce un corrispondente insieme di oggetti ricostruiti (***RecObj***).

Le unità di ricostruzione trattano:

- dati reali forniti dal sistema di acquisizione dati (oppure dati di simulazione);
- oggetti prodotti da altre unità di ricostruzione;
- dati di ambiente come la descrizione del rivelatore, le calibrazioni, ed i parametri che devono essere specificati agli algoritmi di ricostruzione.

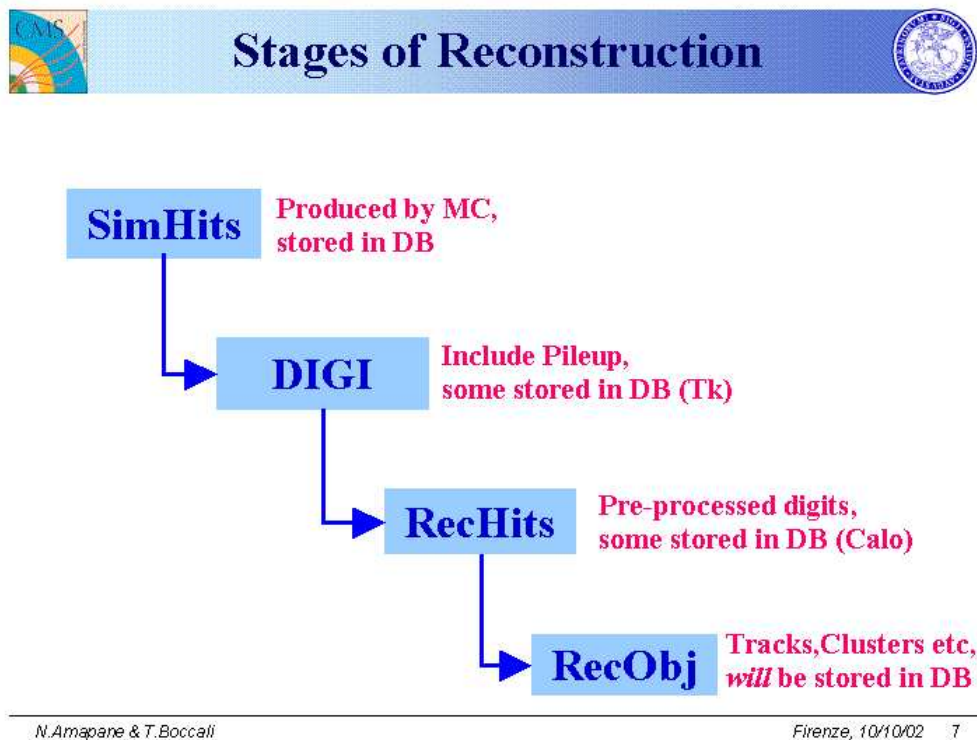

Figura 3.6: I quattro passi per necessari per la ricostruzione di un evento.

3.5.3 Le classi principali utilizzate nel framework

Le principali classi che intervengono nel processo di ricostruzione degli eventi per il modello ad oggetti dell'analisi dei dati sono:

- **RecObj:** classe astratta[5] dalla quale ereditano tutti gli oggetti che apparterranno al modello dei dati ricostruiti per l'analisi fisica;
- **RecUnit:** la classe astratta che rappresenta l'unità di ricostruzione mediante la quale è possibile ottenere gli oggetti ricostruiti (RecObj) per l'analisi fisica finale;
- **RecCollection:** una classe che rappresenta le collezioni di oggetti ricostruiti. Ricordando che ogni oggetto può essere creato a partire da altri oggetti, tale classe costituisce un supporto alla classe RecUnit che si occupa della ricostruzione degli oggetti RecObj;
- **RawData:** particolari sottoclassi di RecObj adoperate per la memorizzazione delle informazioni provenienti dal rivelatore, a partire dalla quale verranno creati i RecHits e altri tipi di RecObj (parag. 3.5.2);

Inoltre COBRA fornisce il supporto per rendere gli oggetti ricostruiti persistenti tramite l'utilizzo di classi "speciali" (TRecRef, TRecVec) attraverso le quali è possibile scrivere e leggere oggetti dal disco. Mentre un puntatore "tradizionale" consente di avere un riferimento ad oggetti (o dati) presenti esclusivamente nella memoria "volatile", l'utilizzo di un'oggetto "TRecRef" in luogo di un tradizionale puntatore consente di avere un riferimento ad oggetti scritti sul disco; in maniera analoga TRecVec rappresenta un vettore di oggetti TRecRef, e quindi un vettore di puntatori ad oggetti persistenti.

"RecUnit" costituisce la classe di base per gli algoritmi di ricostruzione, ogni classe che si occupi della ricostruzione dovrà quindi ereditare da tale classe per essere "consistente" con il framework, e per poter usufruire dei servizi che il framework stesso offre (es: supporto alla persistenza di oggetti). Come già detto *"RecUnit"* può usufruire delle collezioni di oggetti

[5] Se una classe è astratta vuol dire che non possono esistere oggetti apparteneti a quella classe, ma solo oggetti che saranno specializzazioni tramite ereditarietà di quella classe [9].

(ricostruiti) messe a disposizione dalla classe ***"RecCollection"***, ed in in particolare diverse combinazioni delle classi *RecUnit* / *"RecCollection"* possono fornire differenti versioni dello stesso ***"RecObj"***. Un *RecObj* è la classe di base dalla quale ereditano gli oggetti dell'analisi come le tracce, gli elettroni, i jet, etc. Quando l'utente richiede una collezione di oggetti ricostruiti con un algoritmo di oggetti di tipo A, COBRA controlla se quella collezione di oggetti esiste già; se così non è, la ricostruisce utilizzando la ***RecUnit*** che implementa l'algoritmo di ricostruzione di tipo A ("*RecUnit* A"). Se questo algoritmo ha bisogno di altre collezioni di oggetti, ciò indurrà l'invocazione delle corrispondenti RecUnit. In caso di ambiguità tra più algoritmi possibili per il tipo A, l'ambiguità viene risolta introducendo una chiave, che può essere una semplice stringa o un'oggetto più complesso (RecQuery). Una tipica applicazione di questo processo è la ricostruzione del *RecObj* con una stessa *RecCollection* ma usando differenti calibrazioni, allineamenti o parametri per gli algoritmi di ricostruzione.

3.6 I Jet

Nelle collisioni protone–protone nell'acceleratore LHC sono spesso prodotti quarks e gluoni. Secondo la teoria QCD (Quantum Chrono Dynamics) [8] tali particelle non sono osservabili allo stato libero, poichè nel momento stesso in cui vengono prodotte si innesca un processo detto di "frammentazione" attraverso il quale, a partire da queste particelle primarie, sono prodotti nuovamente quarks e gluoni che si ricombinano fino a formare nuove particelle stabili chiamate mesoni e barioni, ovvero adroni (particelle secondarie). Le particelle secondarie provenienti dalla frammentazione hanno una direzione prossima a quella delle particelle originali; inoltre, il principio di conservazione della quantità di moto consente di calcolare con buona approsimazione il momento delle particelle primarie provenienti dalla collisione protone–protone semplicemente sommando i momenti delle particelle secondarie prodotte durante la frammentazione (delle particelle primarie). Pertanto allo scopo di poter ricostruire direzione e momento di quarks e gluoni prodotti dalle collisioni protone–protone, è necessario raggruppare tutte le

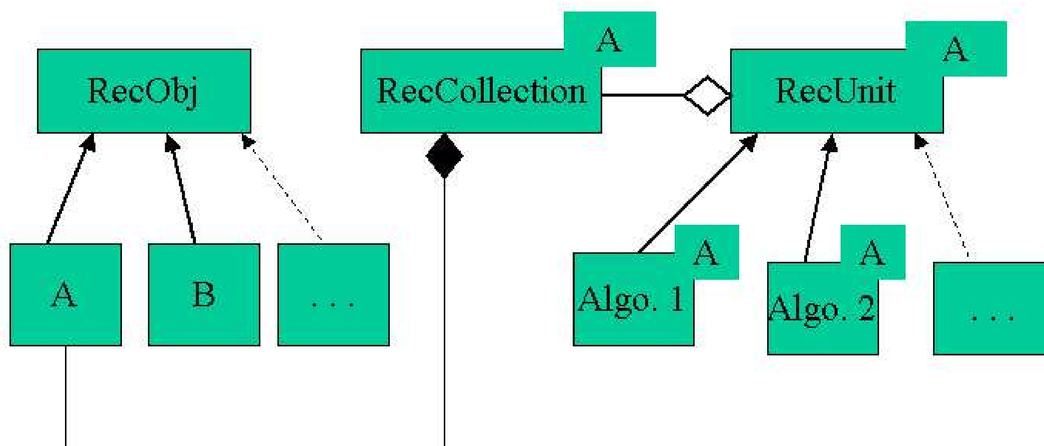

Figura 3.7: Le classi per la ricostruzione degli eventi.

particelle in "*jet*". Un *jet* è quindi l'insieme delle particelle provenienti dalla frammentazione di un singolo partone (quark o gluone). Le particelle nei *jet* sono principalmente adroni e vengono rilevati all'interno del calorimetro andronico (H-CAL). Nella pratica esistono vari algoritmi di ricostruzione dei *jet*, di seguito riporteremo alcuni dei più utilizzati. Anche se in questa tesi non sono stati sviluppati algoritmi di ricostruzione dei *jet*, vengono riportati per completezza le descrizioni di alcuni algoritmi più usati negli esperimenti.

3.6.1 Algoritmi basati sui Coni

Gli algortimi basati sui coni vengono utilizzati soprattutto per collisori adronici (come LHC), ed in alcune applicazioni di collisione elettrone–elettrone. Sono in grado di distinguere tra particelle provenienti da eventi interessanti[6] da particelle dovute ad eventi di *minimum-bias* aventi un basso valore del momento trasverso (impulso). Ciò vuol dire che non tutte le particelle trovate andranno a formare i jet. Nella figura 3.8 in cui le particelle prodotte sono rappresentate tramite frecce colorate viene illustrato chiarmente come alcuni gruppi di particelle (frecce gialle e blu) vadano a costituire dei jet, mentre altre (frecce rosse), non costituiranno un jet poichè provenienti da eventi "non interessanti".

Si parte da un'insieme di depositi di energia significativi (*seeds*) aventi un valore energetico superiore ad una soglia prefissata; Si seleziona il *seed* più energetico e si costruisce un cono di apertura predefinita intorno ad esso; dopodichè si combinano al più energetico tutte le particelle all'interno del cono. Sulla base dei nuovi seed aggiunti si calcola il baricentro, attorni al quale si costruisce un nuovo cono e la procedura viene iterata fino a quando non si trovano più *seeds*) da aggingere al cono. A questo punto, l'algortitmo, tra i seeds non ancora utilizzati, seleziona nuovamente il seed più energetico e procede in maniera analoga cercando le componenti di un nuovo *jet*. Tale processo iterativo termina quando la lista dei depoditi di energia significativi risulta esaurita. La lista dei coni trovati verrà potrà essere inviata ad un'algoritmo che si occuperà, se necessario, di fondere due *jet* eccessivamente piccoli in un unico *jet*, o viceversa scindere un jet in due jet secondo cri-

[6]eventi dovuti alla forza nucleare forte

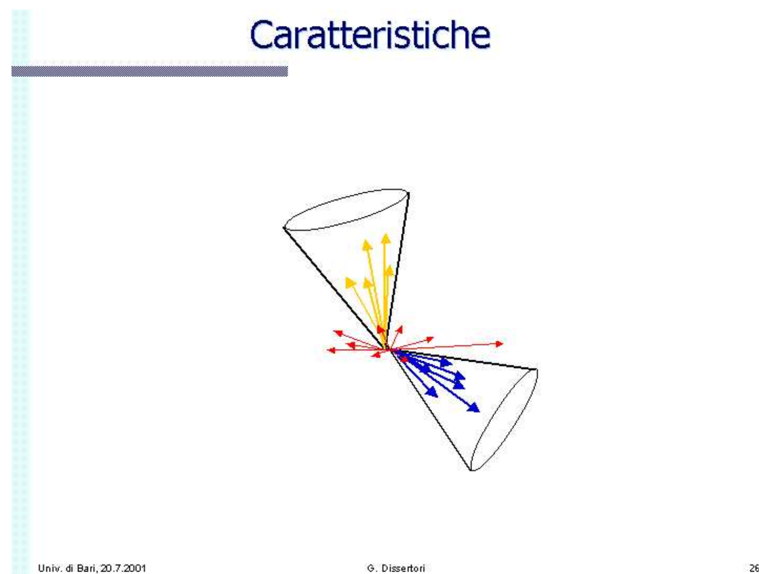

Figura 3.8: Le frecce colorate rappresentano le particelle provenienti dalle collisioni; in particolare le frecce in rosso indicano le particelle provenienti da eventi di scarso interesse e pertanto verranno scartate dagli algoritmi per i jet basati sui coni.

teri che possono variare a seconda dell'implementazione. L'insieme dei coni rimanenti rappresenta la lista di *jet* trovati.

3.6.2 Algoritmi di tipo "Jade"

Al contrario degli algoritmi basati sui coni, gli algoritmi di tipo "JADE", vengono adoperati per lo più negli acceleratori elettrone–elettrone, e più raramente in quelli adronici; al termine di un'algoritmo JADE tutte le particelle faranno parte di un jet. La strategia di raggruppamento è la seguente: due particelle vengono combinate (*clustered*) in una nuova pseudoparticella se risulta $y_{ij} < y_{cut}$, dove y_{ij} è la massa invariante normalizzata y_{cut} rappresenta un valore pre-definito. y_{ij} è definito come:

$$y_{ij} = \frac{2E_iE_j(1 - cos\theta_{ij})}{E^2_{cm}} \approx \frac{m_{ij}}{E^2_{cm}}$$

.

Lo schema di figura 3.9 mostra il diagramma di flusso dell'algoritmo. L'algoritmo quindi procede come segue:

1. si calcola la massa invariante (y_{ij}) per ogni coppia i,j;
2. si memorizza il valore minimo di y_{ij} su tutte le coppie di i,j trovato, nella variabile y_{kl};
3. se $y_{kl} < y_{cut}$:
   - si combinano le componenti k,l;
   - si sommano i loro momenti $P_{kl} = y_k + y_l$;
   - si cancellano k,l dalle liste di costituenti disponibili;
   - infine, si ritorna al passo 1, tranne $y_{kl} >= y_{cut}$; in questo caso l'algoritmo termina.

Una variante dell'algoritmo Jade è rappresenta dall'algoritmo Durham che si differenzia da quest'ultimo solo nella formula per il calcolo di y_{ij}:

$$y_{ij} = \frac{2min(E^2_i, E^2_j)(1 - cos\theta_{ij})}{E^2_{cm}}.$$

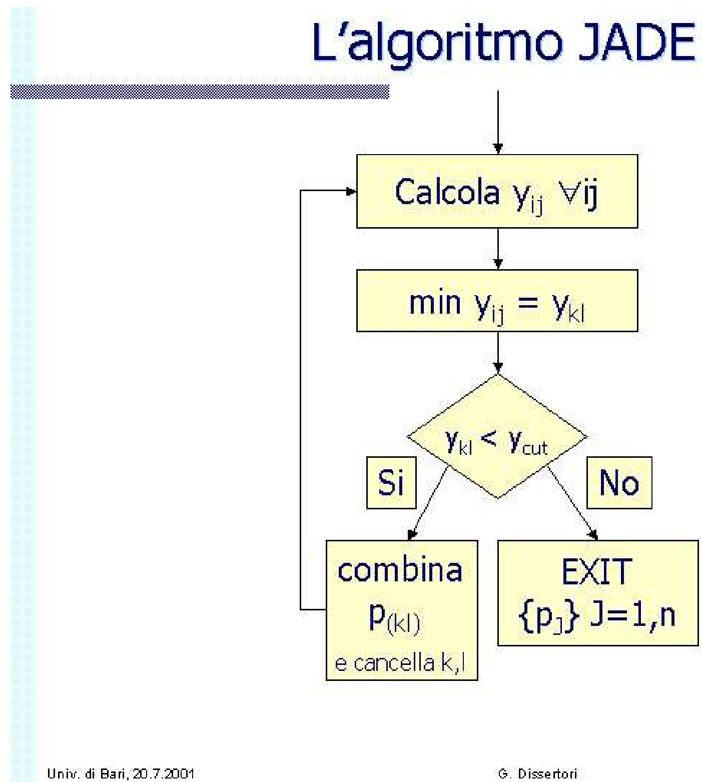

Figura 3.9: Algoritmo di tipo JADE.

Capitolo 4

La persistenza degli oggetti per l'analisi. Studio di un caso particolare: i *"jet"*

.

4.1 Introduzione

Gli studi di fisica in CMS si basano su un modello dei dati, costituito da oggetti ricostruiti a partire dalle informazioni fornite dai sottorivelatori. Per poter effettuare l'analisi fisica finale è necessario che tutti gli oggetti utili appartenenti al modello dei dati siano memorizzati in maniera persistente su un dispositivo di massa oppure siano ricalcolabili da dati persistenti. La scelta di quali collezioni di oggetti scrivere sul disco e quali ricalcolare è legata ad un bilancio in termini delle risorse di CPU e dello spazio di memoria (di massa) richiesto per quel particolare tipo di oggetti. Più semplicemente, se una collezione di oggetti occupa una grossa quantità di spazio sul disco (con conseguente elevato tempo richiesto per le operazioni di input/output), ma richiede tempi di calcolo nettamente inferiori, è preferibile scegliere di ricostruire gli oggetti quando occorrono piuttosto che memorizzarli permanentemente sul disco.

L'obbiettivo di questa tesi è dare un contributo al lavoro necessario per rendere persisteti gli oggetti della ricostruzione allo scopo di fornire un modello ad oggetti dei dati per l'analisi. In particolare, è stato realizzato un prototipo di *"jet persistenti"* per l'analisi. I *jet* risultano un esempio di oggetti persistenti che richiedono la navigazione attraverso i loro costituenti, anch'essi persistenti. Questo rappresenta un esempio abbastanza generale per le problematiche che si riscontrano nell'analisi di dati, per cui soluzioni simili potranno essere adottate da gran parte degli altri oggetti necessari per l'analisi. Al momento in cui questo lavoro di tesi è iniziato, circa quattro mesi fa, esisteva un'architettura software in grado di ricostruire i *jet*, ma non di rendere tali oggetti persistenti.

Il lavoro di tesi è stato quindi articolato nelle seguenti fasi:

1. analisi dell'architettura dei *jet* esistente;
2. revisione di tale architettura in vista della persistenza (in questa fase si riscontrerà che sono necessarie modifiche inaspettate alla struttura preesistente);
3. sviluppo: implementazione delle classi provenienti dalla fase di analisi, dei file necessari per la persistenza, uso dei *template* in C++;
4. test finale delle funzionalità del codice realizzato.

4.2 Fase di Analisi

Prima di procedere nello studio dell'architettura dei *jet* esistente, occorre studiare i servizi di persistenza messi a disposizione dal framework COBRA e quindi, quali sono le operazioni da compiere, e quali i vincoli da rispettare per poter implementare una generica classe persistente.

Il framework di ricostruzione ed analisi degli oggetti dell'esperimento, consente di scrivere classi persistenti senza l'obbligo di definire apposite procedure di Input/Output dal disco. COBRA sfrutta POOL [18], un pacchetto realizzato appositamente per la scrittura e la lettura di oggetti dal disco.

POOL è un pacchetto software realizzato al CERN nell'ambito del progetto LCG (LHC Computing Grid) [?], la cui attuale implementazione si basa su un'altro prodotto sviluppato al CERN, denominato ROOT [14] che opera a più basso livello per la scrittura degli oggeti su disco. Le operazioni fondamentali da compiere per poter scrivere classi persistenti usufruendo di COBRA sono di seguito elencate:

1. Le classi degli oggetti ricostruiti che devono essere scritti sul disco devono ereditare dalla classe (astratta) *RecObj* (vedi 3.5.3);
2. Bisogna specificare al pacchetto di I/O (POOL) il "catalogo" delle classi da da rendere persistenti. L'elenco di tali classi andrà descritto in un apposito file attraverso il meta–linguaggio XML; oltre ai nomi delle classi da scrivere sul disco, all'interno del file è possibile, opzionalmente, specificare gli attributi della classe che non necessitano di essere memorizzati in maniera permanente (esempio: variabili usate solo come *cache*, ma che non fanno parte dello stato dell'oggetto in C++ definito come: "*mutable*"); I dettagli implemetativi di questo procedimento sono riportati nella sezione 4.3 relativa allo sviluppo del codice.
3. Ogni oggetto che eredita dalla classe *RecUnit* (3.5.3) deve essere passato come parametro alla classe parametrica (template) chiamata "RecBuilder". Attraverso questa classe sarà possibile (impostando opportunamente un file di configurazione usato a run–time) sia leggere che scrivere oggetti dal disco.

Alla luce di quanto detto osserviamo criticamente il modello delle classi dei *jet* esistente mostrato in figura 4.1.

4.2.1 Prima iterazione sul *design*

La classe *RecObj* rappresenta la classe da cui tutti gli oggetti ricostruiti devono ereditare (3.5.3); le classi (astratte) "VJet" e "VJetableObject" costituiscono le interfacce (astratte) rispettivamente della classe "ConcreteJet" (la classe dei *jet*), e "JetableObject" (la classe degli oggetti che costituiscono un *jet*). "JetableObject" è una classe parametrica, realizzata in C++

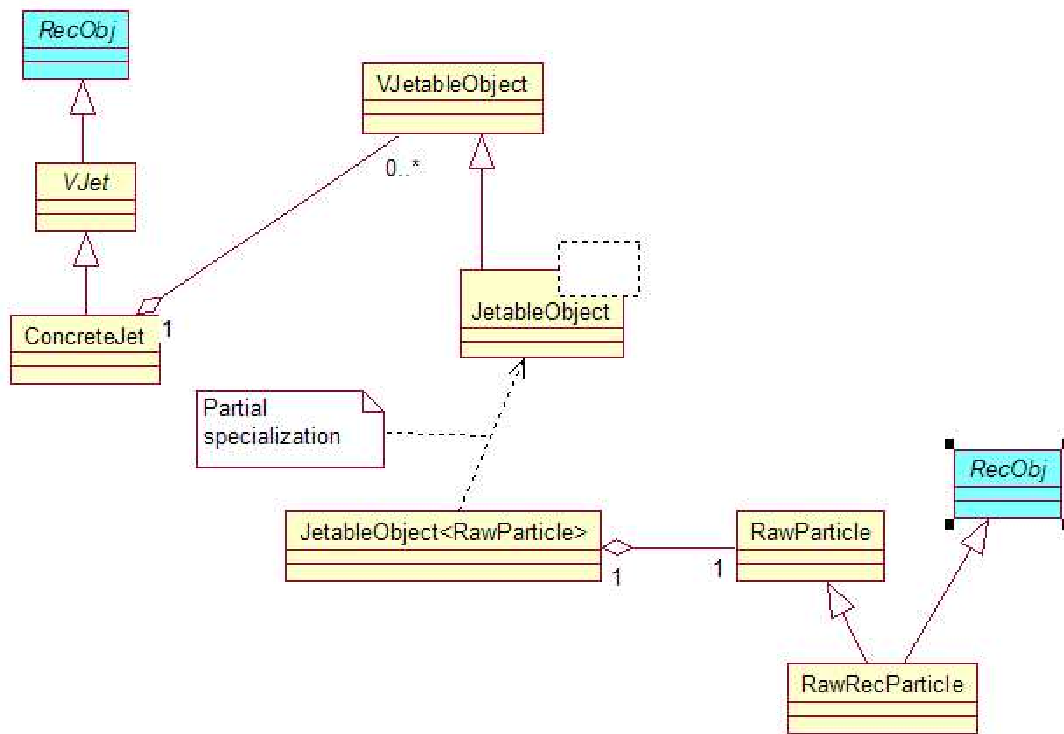

Figura 4.1: UML dell'architettura dei *jet* corrente.

utilizzando un *template* (vedi: sez. 4.3, bibliofrafia [9]); “JetableObject$< RawParticle >$” rappresenta una specializzazione tramite la classe “RawParticle” di “JetableObject”. La classe“RawParticle” costituisce la classe più elementare per l’identificazione di una particella, ha un solo attributo costituito da un quadri-vettore di Lorentz, necessario per memorizzare il momento associato ad una generica particella all’interno dell’esperimento, ed è utilizzata nei test attuali poichè fornisce un esempio semplice di costituenti di *jet*. Lo stesso software potrà essere poi utilizzato senza alcuna modifica, su altri tipi di costituenti (es: cluster calorimetrici, tracce, etc.) semplicemente utilizzando un tipo diverso di oggetti come parametro *template* della classe JetableObject. “RawRecParticle” è una particolare sottoclasse persistente di RawParticle; occorre inoltre precisare che la classe “RawParticle” pur essendo utilizzata all’interno di questo diagramma delle classi per i *jet* appartiene (insieme con “RawRecParticle”) al framework di CMS, COBRA, e pertanto risulta possibile sfruttare questa classe, ma non è consentito apportarvi alcuna modifica.

A questo punto sulla base dei requisiti per la persistenza richiesti dal framework, la prima incompatibilità che si riscontra è che la classe “VJetableObject” non viene gestita dal framework dell’esperimento (COBRA), bensì dal modulo di ricostruzione mostrato in figura 4.2 che fa uso del *patterns “facade”* [22]. Attraverso il *patterns-facade* tutte le classi del framework di ricostruzione dei *jet* risultano nascoste alla *RecUnit*, ovverro la comunicazione tra COBRA e le classi della ricostruzione dei jet può avvenire soltanto se mediata dalla “facciata”,rappresentata dalla classe “JetFinderCARFFacade”. Questo vuol dire che l’attuale modulo di ricostruzione dei jet, risulta totalmente indipendente da COBRA, pertanto per poter adattare l’attuale struttura transiente ad una nuova architettura persistente occorrerà sostituire la classe VJetableObject con una classe in grado di essere gestita da COBRA, e si dovrànno apportare sostanziali modifiche al corrente modulo di ricostruzione.

Analizziamo quindi le classi della ricostruzione (Fig. 4.2), in vista di una nuova struttura in grado di sfruttare i servizi di persistenza offerti dal framework.

Le classi VInputCalibrator e VJetCalibrator si occupano della calibrazio-

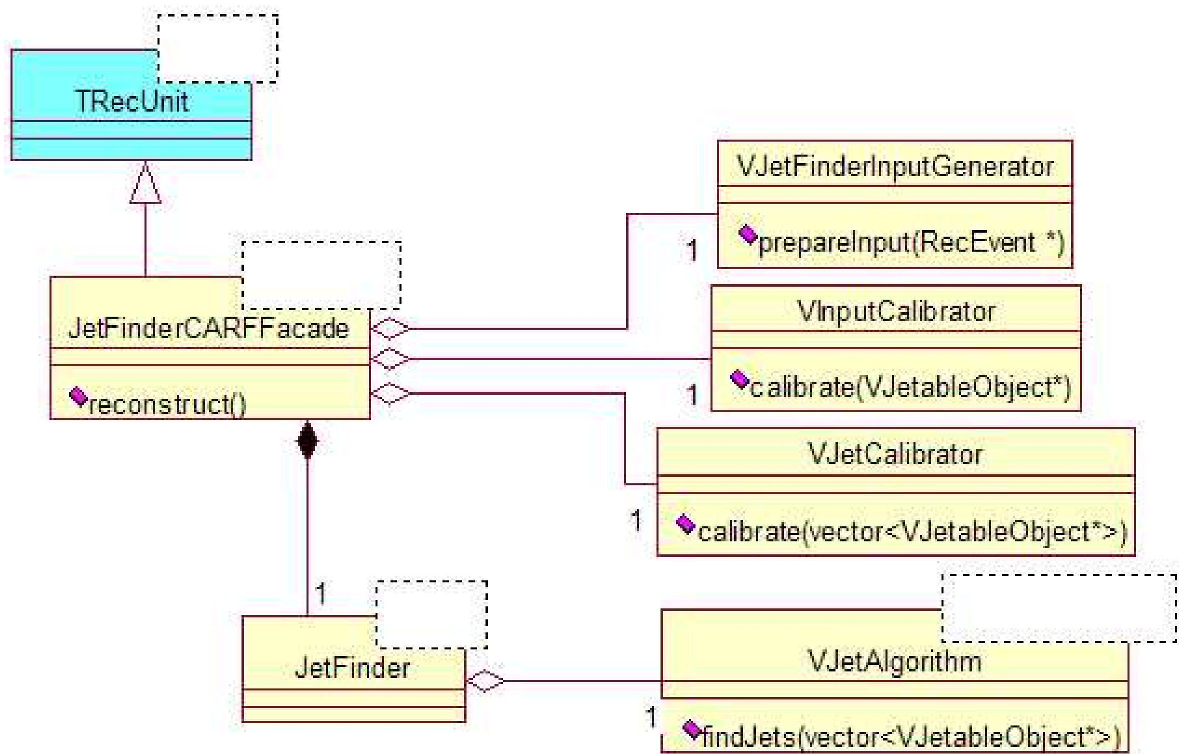

Figura 4.2: Il framework di ricostruzione dei *jet* (transiente).

ne[1] rispettivamente della classe dell'input e per quella dei *jet*. VJetAlgorithm è la classe di base degli algoritmi di ricostruzione dei *jet* i cui oggetti sono gestiti attraverso la classe VJetFinder.

La classe più interessante ai fini della persistenza dei *jet* è la classe "VJet-FinderInputGenerator" che fornisce tramite il metodo "prepareInput()" i costituenti della classe dei *jet*, i VJetableObject (Fig. 4.1), che dovranno essere poi passati all'algoritmo di ricostruzione implementato nella classe JetFinder. Le attuali classi generatrici dell'input dovranno essere sostituite con dei moduli COBRA, che erditano dalla classe *RecUnit*; diversamente non risulterebbe possibile scrivere sul disco i costituenti dei *jet* utilizzando i servizi del framework. Sulla base di tali considerazioni, si è giunti ad un primo prototipo di architettura per i *jet* persistente, mostrato nell'UML [13] riportato in figura 4.3.

Al fine di ridurre l'impatto delle modifiche sul codice esistente, sono state aggiunte nuove classi (in verde) "*jet*", "*JetCostituent*", e "*Jet<JetCostituent>*" senza apportare cambiamenti alle classi esistenti. Al termine della migrazione verso la nuova struttura peristente, le vecchie classi potranno essere eliminate. La nuova classe *jet* ha sostazialmente le stesse funzionalità (metodi) e gli stessi attributi della classe *ConcreteJet* , ma è stata adattata alla persistenza sfruttando una delle classi (TRecVec) messe a disposizione dal framework COBRA (4.3 per i dettagli) per consentire la navigazione attraverso i costituenti persistenti. TRecVec rappresenta un vettore di "oggetti puntatori" implementati tramite la classe TRecRef. L'utilizzo di "TRecRef" in luogo di un tradizionale puntatore[2] consente di avere un riferimento ad oggetti scritti sul disco; di conseguenza la classe "TRecVec" rappresenta un vettore di puntatori ad oggetti persistenti. Analogamente *JetCostituent* rappresenta la versione persistente di *JetableObject*. La nuova classe *JetCostituent* eredita da *RecObj* rispettando così i vincoli imposti da COBRA, inoltre non presenta,

[1] Il problema della della calibrazione e della persitenza dei relativi oggetti, non è trattato in questo lavoro, ma la nuova architettura proprosta consetirà di implementare la persistenza degli oggetti della calibrazione senza troppe difficoltà utilizzando servizi del framework attualmente in fase di sviluppo.

[2] Ricordiamo che l'utilizzo di puntatori tradizionali consente di avere un riferimento ad oggetti (o dati) presenti esclusivamente nella memoria "volatile".

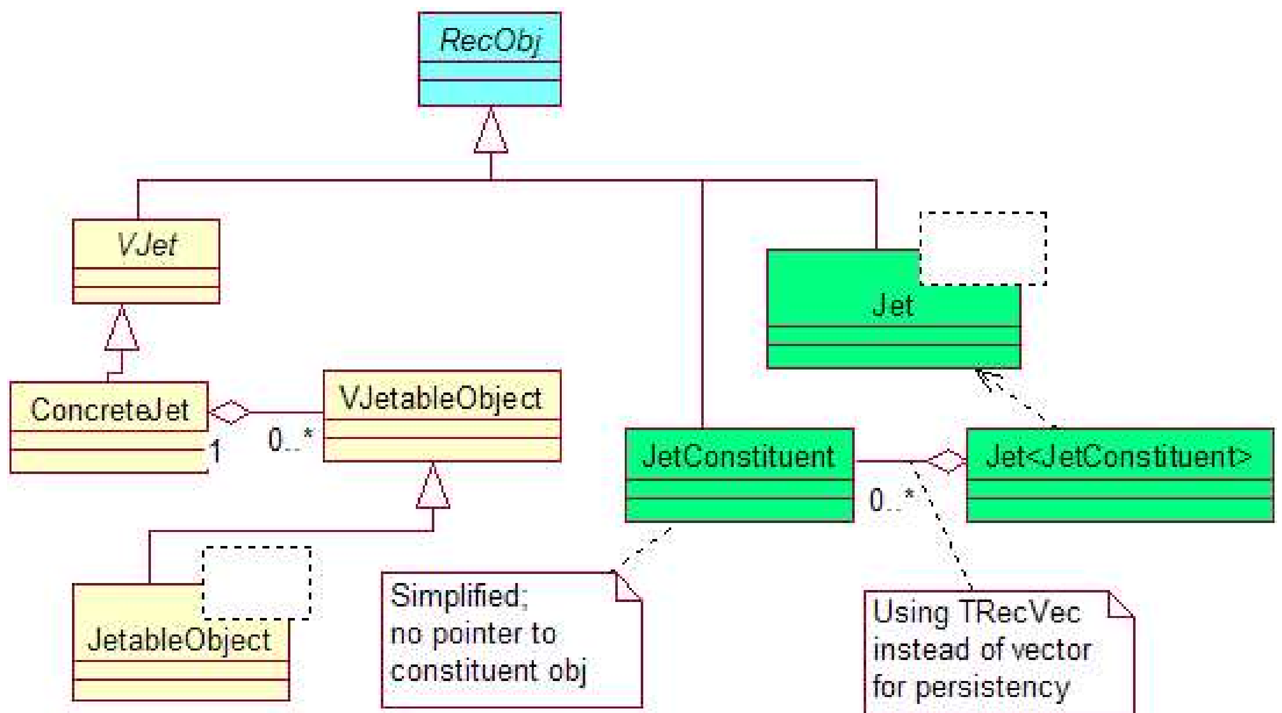

Figura 4.3: UML delle classe per la rappresentazione di *jet* persistenti (I versione).

per il momento, il puntatore alla classe RawParticle, che sarà introdotto nelle fasi successive di iterazione sul *design*; gli attributi di quest'ultima classe sono direttamente memorizzati all'interno del nuovo oggetto JetCostituent. Con questo approccio si evita di trattare la persistenza di un'ulteriore oggetto. La classe *Jet<JetCostituent>* è una specializzazione tramite *template* della classe dei *jet*, ossia è possibile costruire nuovi *jet* a partire da nuove classi di costituenti semplicemente passando l'eventuale nuova classe come argomento del template in luogo dell'attuale *JetCostiuent*. L'uso del template piuttosto che l'uso della classe base VJetableObject evita anche la necessità di effettuare "*cast*" [9] per navigare attraverso i costituenti. Il *cast* infatti costituisce una "violazione" del polimorfismo.

La relazione tra le classi Jet<JetCostituent> e JetCostituent è una relazione di aggragazione "uno a molti", nel senso che "un *jet*" può essere composto da "molti costituenti".

Anche gli algoritmi dei *jet* sono stati riscritti come classi parametriche ("template") essendo il tipo dei *jet* passato come parametro del *template*. Inoltre, la classe *JetCostiuent* potrà in seguito essere sostituita da classi più complesse semplicemente passando l'eventuale nuova classe come argomento del template in luogo dell'attuale *JetCostiuent*. Ovvero, la flessibilità apportata dai template consente di costruire oggetti *jet* fatti di altri oggetti omogenei concreti, o in alternativa, *jet* costituiti da oggetti costituenti eterogenei, ponendo come argomento del template la classe di base della quale ereditano i costituenti concreti.

4.2.2 Seconda iterazione sul *design*

Dopo aver implementato e testato l'architettura proprosta con un certo numero di eventi simulati forniti da OSCAR (sez. 3.4), è stata fatta un seconda fase di *design* attraverso la quale si è giunti alla nuova architettura mostrata in figura 4.4. Questa seconda iterazione sul *design* ha portato ad una struttura ancora più generale e quindi più flessibile. È stata trasformata in *template* anche la classe *JetCostituent*, e sono state aggiunte nuove classi: "*JetCostituentBase*", che rappresenta la classe base da cui i costituenti erediteranno, e "*JetCostituent<RawRecParticle>*" che adesso ha un riferimento alla classe

"*RawRecParticle*" implementato con l'ausilio della classe "TRecRef" fornita da COBRA.

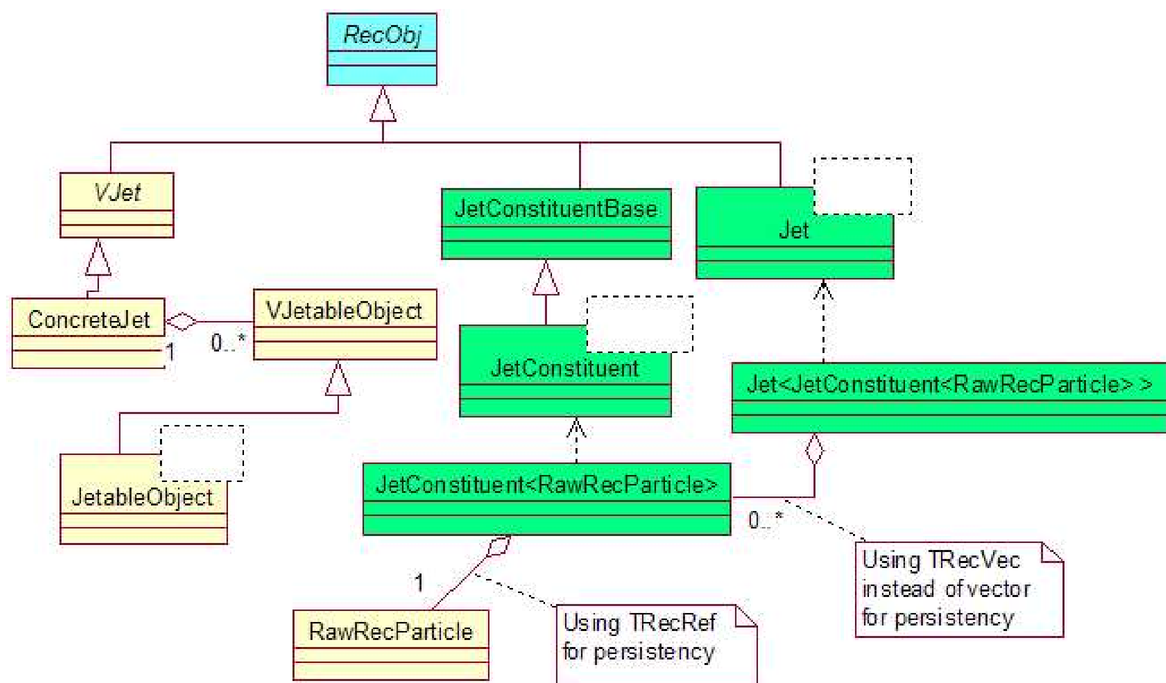

Figura 4.4: UML delle classi per la rappresentazione di *jet* persistenti (II versione).

Per quanto riguarda le modifiche al framework di ricostruzione dei *jet* sulla base dei problemi riscontrati nell'approccio *facade* preesistente (fig. 4.2), e in linea con le richieste di COBRA per implemetare classi persistenti, la nuova architettura di ricostruzione illustrata in figura 4.5 mostra che sia la classe per la ricostruzione dei JetCostituent (JetCostituentBuilder), che quella per la ricostruzione dei Jet (JetBuilder) attualmente ereditano da *RecUnit* in accordo con i requisiti richiesti per la persistenza.

Essendo il nostro obbiettivo la persistenza dei *jet* e non la loro ricostruzione, la struttura per l'implementazione degli algoritmi è sostanzialmente simile a quella preesistente: una classe astratta VJetAlgorithm dalla quale ereditano i vari algoritmi di ricostruzione (es: IterativeConeAlgorithm), e il

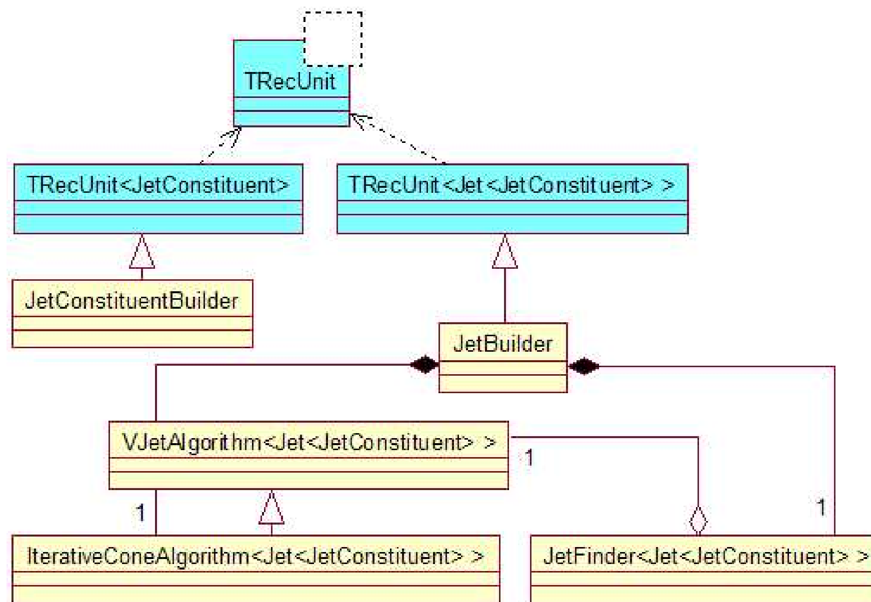

Figura 4.5: Il framework di ricostruzione dei *jet* persistente (II versione). Diverse altre classi ausiliarie, non mostrate nei diagrammi UML quì riportati, necessarie nel precedente modello, possono ora essere eliminate perchè non più necessarie nel nuovo modello.

JetFinder che non fa altro che riempire un vettore di costituenti ed applicare l'algoritmo scelto[3].

4.2.3 Terza iterazione sul *design*

Nonostante ciò, un'ulteriore semplificazione è possibile implementando direttamente l'algoritmo di ricostruzione ("Algoritmo X") come modulo che eredita direttamente da *RecUnit* ("JetBuilder_Algoritmo X"). In effetti, lo scopo del "Builder–Patterns" [22] è differenziare la rappresentazione di un'oggetto dalla sua ricostruzione; in tal modo è possibile avere una stessa rappresentazione di un'oggetto utilizzando strategie di ricostruzione differenti (e viceversa); pertanto in questo caso è consigliabile implementare direttamente diversi JetBuilder (es: JetBuilder_IterativeCone, JetBuilder_SimpleConeAlgorithm, etc.) che ereditano da una classe base JetBuilderBase[4], in luogo della struttura di ricostruzione che adopera VJetAlgorithm e JetFinder. Anche questa ulteriore semplificazione è stata implementata e testata con successo e il grafico in notazione UML è mostrato in figura 4.6. (Vedi appendice A per dettagli).

[3]il tipo di algoritmo da eseguire viene passato come parametro ad un apposita funzione del JetFinder.

[4]Che ovviamente deve ereditare da TRecUnit per poter essere in linea con le richieste del framework COBRA ed usufruire quindi del supporto alla persistenza.

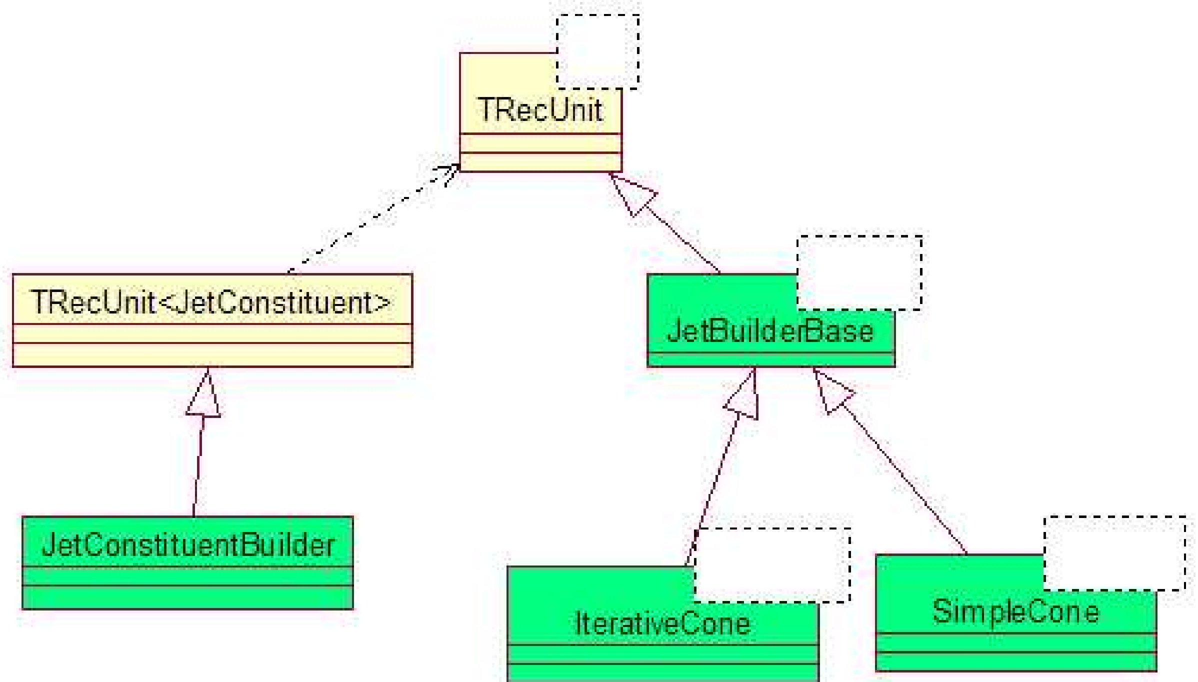

Figura 4.6: Il framework di ricostruzione dei *jet* persistente (III versione).

4.3 Fase di Sviluppo

In questa sezione dedicata alla fase di sviluppo del codice, verranno esaminati le parti fondamentali per la persistenza e alcuni aspetti di programmazione generica. Iniziamo riportando un'esempio, relativo al caso specifico dell'architettura dei *jet*, di catalogo di classi (particolari) persistenti (scritto in XML).

```
<lcgdict>
  <class name="JetConstituentBase">
      <field name="used" transient="true"/>
  </class>

  <class name="JetConstituent<RawRecParticle>"/>

  <class name="TRecRef<RawRecParticle>"/>

  <class name="Jet<JetConstituent<RawRecParticle> >">
      <field name="theEta" transient="true"/>
      <field name="thePhi" transient="true"/>
      <field name="theTheta" transient="true"/>
      <field name="theEnergy" transient="true"/>
      <field name="theMass" transient="true"/>
      <field name="the4Vector" transient="true"/>
      <field name="done" transient="true"/>
  </class>


  <class name="TRecVec<JetConstituent<RawRecParticle> >">
      <field name="trec" transient="true"/>
  </class>
</lcgdict>
```

Come si nota dal codice riportato, alcune classi (JetConstituentBase, Jet<JetConstituent<RawRecParticle>>, TRecVec< JetConstituent<RawRecParticle> >) hanno parte dei loro attributi (*field name* = "attributo") seguiti dalla *flag* *"transient"* impostata a "*true*". Questa *flag* serve a specificare quale attributo della classe non necessita di essere scritto sul disco, ossia quale attributo deve rimanere transiente. Ad esempio le coordinate del quadri–momento possono essere ricalcolate a partire dai costituenti, e le variabili "thePhi", "thetheta", etc. sono usate solo per memorizzare in una *cache* il valore dell'operazione.

Osserviamo inoltre la differenza tra gli attributi della classe dei *jet* transiente (ConcreteJet), gli attributi della nuova classe adattata alla persistenza (Jet):

Attributi della classe dei *jet* preesistente

```
private:

 // buffered values
    double theEta;
    double thePhi;
    double theTheta;
    double theEnergy;
    double theMass;
    HepLorentzVector the4Vector;

// status flag of the buffering
     int done;

// list of costiutents
  std::vector<const VJetableObject *> theConstituants;
```

Attributi della nuova classe dei *jet* persistente

```
private:
```

```
// buffered values
   mutable double eta;
   mutable double phi;
   mutable double theta;
   mutable double e;
   mutable double m;
   mutable HepLorentzVector p4;

// status flag of the buffering
   mutable int done;

// list of costituents

   container collection;
%};
```

dove "container" è definita attraverso un typedef come segue: `typedef TRecVec<constituent> container;` e `''costituet''` rappresenta il parametro passato come *template* alla classe Jet; in sostanza `costituent` rappresenta il tipo degli oggetti che costituiscono il Jet. Le variabili `eta, phi, theta, ...` rappresentano solo valori di *cache* che corrispondono al momento totale del *jet*, calcolabile a partire dai costituenti. Quindi la relazione di aggragazione tra un *jet* e i suoi costituenti rappresentata attraverso un vettore (transiente) di VJetableObject (std::vector<const VJetableObject $*$ > theConstituants) è stata ora sostituita dalla definizione `''TRecVec<constituent> collection''`; questa definizione sfrutta la classe (templatata) `TRecVec<T>` messa a disposizione dal framework che consentirà di avere un vettore di puntatori a oggetti persistenti (in luogo del vettore di puntatori ad oggetti transienti).
In maniera analoga il riferimento ad un'oggetto transiete presente come attributo della classe JeableObject è stato sostituito da un riferimento ad un'oggetto persistente con l'ausilio della classe `TRecRef<T>` fornita da COBRA;

di seguito verranno riportati i codici delle classi JetableObject (prima) e JetCostituent (dopo) allo scopo di mostrare le differenze tra le due classi. Il codice di *JetCostituent* si riferisce alla prima versione del *design* proposto per cui non vi sono ancora sostanzili modifiche rispetto al codice preesistente.

JetableObject.h

```
#ifndef JetableObject_h
#define JetableObject_h

#include "Jets/AbstractJet/interface/VJetableAdapter.h"
#include "Jets/AbstractJet/interface/TJetableAdapter.h"
#include "Jets/AbstractJet/src/VJetableObject.h"

//The object to find jets on - will be adapted to for
// the individual use-cases

template <class ConType>

class JetableObject : public VJetableObject
{
public:
  JetableObject(const ConType & aConstituent);

  const ConType *getConstituent() { return theConstituent; };

private:
// no responsibility for deleting theConstituent.
  const ConType  *theConstituent;
};
#include "Jets/JetFramework/src/JetableObject.icc"
#endif
```

JetCostituent.h

```
#ifndef JetConstituent_h
#define JetConstituent_h

#include "CARF/Reco/interface/RecObj.h"
#include "Jets/AbstractJet/src/VJetableObject.h"

// Written by alex

class RawParticle;

class JetConstituent : public RecObj, public VJetableObject
{
 public:
  JetConstituent( const RawParticle & p );
  JetConstituent();
  ~JetConstituent();
};

#endif
```

Dal punto di vista della programmazione generica occorre dire che in tutto il codice è stato fatto largo uso della potenza dei template. I template rappresentano senza dubbio una delle funzionalità più sofisticate e potenti del C++, aiutano a raggiungere uno degli obbiettivi più sfuggenti della programmazione: la creazione di codice riutilizzabile. Attraverso i template è possibile creare funzioni e classi generiche, nelle quali il tipo di dato su cui esse operano viene specificato sotto forma di parametro [9]. Il codice dei Jet (sotto) (e quello dei JetCostituent relativamente all'ultima versione della classe riportata in appendice) mostrano esempi di utilizzo dei template all'interno della nuova architettura; inoltre essendo il tipo dei *jet* un parametro template è stato possibile estendere questa funzionalità anche alle classi egli

algoritmi di ricostruzione dei *jet*. Nel seguito è riportata l'interfaccia della classe dei Jet.

Jet.h

```
#ifndef Jet_h
#define Jet_h

#include <vector>
#include "COBRA/Reco/interface/TRecRef.h"
#include "COBRA/Reco/interface/RecObj.h"
#include <CLHEP/Vector/LorentzVector.h>
#include <cmath>

template<typename Constituent>
class Jet : public RecObj
{
public:
  typedef Constituent constituent;
  typedef TRecVec<constituent> container;
  typedef typename container::const_iterator const_iterator;

  Jet();
  Jet(HepLorentzVector & aLorentzVector);
  ~Jet();

  double getEta() const;
  double getPhi() const;
  double getTheta() const;
  double getEnergy() const;
  double getMass() const;
  HepLorentzVector getLorentzVector() const;
  int size() const { return collection.size(); }
  //void push_back( const T * t ) { collection.push_back( t ); }
  const_iterator begin() const { return collection.begin(); }
```

```
  const_iterator end() const { return collection.end(); }

  void addConstituent(const constituent * aPart) {
    collection.push_back(aPart); done = 0; }

private:
  container collection;

  void buffer() const;
  mutable int done;  // status flag of the buffering
   // buffered values
  mutable double eta;
  mutable double phi;
  mutable double theta;
  mutable double e;
  mutable double m;
  mutable HepLorentzVector p4;
};
```

Capitolo 5

Conclusioni

Questo lavoro di tesi si è svolto nell'ambito del gruppo di Napoli dell'Istituto Nazionale di Fisica Nucleare (INFN) che partecipa all'esperimento CMS, al CERN (Ginevra), attualmente in fase di costruzione e previsto cominciare ad operare nel 2007. CMS rappresenta un grande rivelatore posto sul più potente acceleratore di particelle mai costruito, il Large Hadron Collider (LHC). LHC sarà in grado di far collidere fasci di protoni a livelli di energie mai raggiunti prima. Il rivelatore CMS servirà ad identificare le particelle prodotte da tali collisioni, misurandone una serie di informazioni come direzione, momento, energia, etc.. Attraverso tali informazioni, e con l'ausilio di un linguaggio d'alto livello (C++), sarà possibile ricostruire un "modello dei dati" ad oggetti sul quale si baseranno gli scienziati per effettuare "l'analisi fisica" finale nella quale si cercherà di scoprire nuove particelle e nuovi fenomeni della fisica alle alte energie, previsti dal Modello Standard Elettrodebole, ma non ancora osservate sperimentalmente.

Il contributo all'eperimento fornito con il lavoro di tesi è stato realizzare un prototipo di oggetti persistenti per il modello dei dati dell'analisi fisica; in particolare si è scelto di lavorare alla classe di oggetti "jet" provenienti dalla ricostruzione.

L'obbiettivo primario è riuscire a scrivere sul disco collezioni di oggetti "jet", ma a causa di un'architettura (transiente) di base poco funzionale e non facilmente adattabile a nuovi requisiti, come quello della persistenza, è stato necessario apportare estese modifiche al design esistente, per poter adattare

l'architettura esistente ad una nuova struttura in grado di essere memorizzata su dispositivi di massa in maniera persistente, utilizzando i servizi forniti dal framework dell'esperimento.

Alla conclusione dell'esperienza maturata sul complesso sistema software dell'esperimento CMS, ho potuto apprendere concetti importanti spesso dimenticati, o sottovalutati da coloro che si apprestano a risolve un problema tramite linguaggi d'alto livello ed in particolare tramite i linguaggi "Object Oriented". Da un lato ho appreso la potenza e i vantaggi di una struttura software ad oggetti basata sui "patterns", come accade all'interno del framework di CMS. Dall'altro ho costatato che non sempre l'utilizzo dei "patterns", se non applicati in maniera "oculata", rappresenta la scelta migliore. Talvolta un'accurato lavoro di analisi può portare ad un'architettura che risulti più snella, più funzionale e quindi complessivamente più efficiente. Nel caso specifico, l'architettura preesistente per l'implementazione dei jet, adottava il pattern "facade". A seguito di uno studio ripartito tra: framework di CMS, corrente struttura delle classi per i jet e per la loro ricostruzione, e "persistenza" nel framework, l'architettura preesitente è risultata poco funzionale e di non facile adattabilità.

La nuova struttura realizzata, oltre ad essere più chiara e quindi semplice da apprendere, è in grado di: gestire la ricostruzione degli oggetti "jet" e la navigazione tra i suoi componenti, fornire la flessibilità di scegliere il tipo (la classe) degli oggetti costituenti un jet, memorizzare su un dispositivo di massa in maniera persistenza le collezioni di oggetti (jet) provenienti dalla ricostruzione di eventi simulati[1]*.*

Un'altra importante lezione appresa è che, nella progettazione del software è spesso utile distinguere se il risultato finale del nostro lavoro rappresenterà un modulo a se stante, oppure (come in CMS) costituirà una componente che dovrà essere integrata all'interno di un framework ad un livello più alto. Nel secondo caso è bene sfruttare al meglio i servizi che il framework mette a disposizione, in modo da evitare lo sforzo necessario per riprodurre queste funzionalità nel proprio codice. Tornando al caso in esame della struttura

[1] Ricordiamo che acceleratore LHC e rivelatore CMS sono in fase di costruzione per cui tutto il software attuale dell'esperimento si basa su dati provenienti dalla simulazione.

dei jet preesistente, uno dei problemi fondamentali del design scelto era costituito dal fatto che il framework di ricostruzione degli oggetti non risultava perfettamente integrato nell'intero framework dell'esperimento (COBRA), e pertanto non era in grado di sfruttare servizi come il supporto alla persistenza che il framework stesso (COBRA) offre. Il nuovo software di ricostruzione e rappresentazione dei jet realizzato è dotato di una struttura molto flessibile, e potrà essere sfruttato come modello per l'implementazione di molti altri oggetti apparteneti al modello dei dati. Considerando inoltre, che molti degli oggetti che apparterranno al modello dei dati per lanalisi fisica finale dovranno essere memorizzati in maniera persistente, risulta ovvia l'importanza di progettare strutture che risultino consistenti con il framework di CMS, e siano facilmente adattabili alla persistenza.

In conclusione il nuovo modello è stato, in seguito alla fase di analisi, progettato, quindi sviluppato in C++ nell'ambito del software dell'esperimento, infine testato nelle sue funzionalità. Ne è stato proposto il rilascio nell'utilizzo del software ufficiale dell'esperimento nella prima release dove sarà possibile incorporare nuovi sviluppi.

Appendice A

Appendice A

In questa appendice sono riportati i sorgenti che descrivono le classi relativamente alla terza versione dell'architettura dei Jet persistente proposta, illustrata nei diagrammi UML di figura 4.4 e 4.6;

Jet.h

```
#ifndef Jet_h
#define Jet_h

#include <vector>
#include "CARF/Reco/interface/TRecRef.h"
#include "CARF/Reco/interface/RecObj.h"
#include <CLHEP/Vector/LorentzVector.h>
#include <cmath>

/*** Written by alex ***/

template<typename Constituent>
class Jet : public RecObj
{
public:
```

```
  typedef Constituent constituent;
  typedef TRecVec<constituent> container;
  typedef typename container::const_iterator const_iterator;

  Jet();
  Jet(HepLorentzVector  &aLorentzVector);
  ~Jet();

  double getEta() const;
  double getPhi() const;
  double getTheta() const;
  double getEnergy() const;
  double getMass() const;
  HepLorentzVector getLorentzVector() const;
  int size() const { return collection.size(); }

  //void push_back( const T * t ) { collection.push_back( t ); }
  const_iterator begin() const { return collection.begin(); }
  const_iterator end() const { return collection.end(); }

  void addConstituent(const constituent * aPart) {
    collection.push_back(aPart); done = 0; }

private:
  container collection;

  void buffer() const;
  mutable int done;  // status flag of the buffering
   // buffered values
  mutable double eta;
  mutable double phi;
  mutable double theta;
  mutable double e;
  mutable double m;
```

```
  mutable HepLorentzVector p4;
};

template<typename Constituent>
Jet<Constituent>::Jet() :
  RecObj(), eta(0), phi(0), theta(0), e(0), m(0), p4(0,0,0,0),
  done( 0 ) {
}

template<typename Constituent>
Jet<Constituent>::Jet(HepLorentzVector & p) :
  RecObj(), phi( p.phi() ), theta( p.theta() ),
  e( p.t() ), m( p.m() ), p4( p ),
  done( 1 ) {
  double pz = p.z();
  eta =  0.5 * log( ( e + pz ) / ( e - pz ) );
}

template<typename Constituent>
Jet<Constituent>::~Jet() {
}

template<typename Constituent>
double Jet<Constituent>::getEta() const {
  buffer(); return eta;
}

template<typename Constituent>
double Jet<Constituent>::getPhi() const {
  buffer(); return phi;
}

template<typename Constituent>
double Jet<Constituent>::getTheta() const {
```

```
  buffer(); return theta;
}

template<typename Constituent>
double Jet<Constituent>::getEnergy() const {
  buffer(); return e;
}

template<typename Constituent>
double Jet<Constituent>::getMass() const {
  buffer(); return m;
}

template<typename Constituent>
HepLorentzVector Jet<Constituent>::getLorentzVector() const {
  buffer(); return p4;
}

template<typename Constituent>
void Jet<Constituent>::buffer() const {
  if( !done && collection.size() != 0 ) {
    done = 1;
    static HepLorentzVector zero(0,0,0,0);
    p4 = zero;
    for( const_iterator i = collection.begin(); i != collection.end(); ++i ) {
      p4 += (*i)->getLorentzVector();
    }
    e = p4.t();
    phi = p4.phi();
    theta = p4.theta();
    m = p4.m();

    double pz = p4.z();
```

```
    eta =  0.5 * log( ( e + pz ) / ( e - pz ) );
  }
}

#endif
```

JetCostituentBase.h

```
#ifndef JetConstituentBase_h
#define JetConstituentBase_h

#include "CARF/Reco/interface/RecObj.h"

// Written by alex
class HepLorentzVector;

class JetConstituentBase : public RecObj
{
 public:
  JetConstituentBase();
  ~JetConstituentBase();
  double getEta() const { return eta; }
  double getPhi() const { return phi; }
  double getTheta() const { return theta; }
  double getEnergy() const { return e; }
  double getTransverseEnergy() const;
  HepLorentzVector getLorentzVector() const;
  int isUsed() const {return used;}
  void use() const { used = 1; }
  void reUse() const { used = 0; }

 protected:
  JetConstituentBase( double, double, double, double );
```

```
  double e;
  double eta;
  double phi;
  double theta;
  mutable int used;
};

#endif
```

JetCostituentBase.cc

```
#include "Utilities/Configuration/interface/Architecture.h"
#include "Jets/PersistentJet/interface/JetConstituentBase.h"
#include <CLHEP/Vector/LorentzVector.h>

JetConstituentBase::
JetConstituentBase( double e_, double eta_, double phi_, double theta_ ) :
  RecObj(), e( e_ ), eta( eta_ ), phi( phi_ ), theta( theta_ ) {
}

JetConstituentBase::JetConstituentBase() :
  RecObj() {
}

JetConstituentBase::~JetConstituentBase() {
}

double JetConstituentBase::getTransverseEnergy() const {
  return e * sin( theta );
}

HepLorentzVector JetConstituentBase::getLorentzVector() const {
  double st = sin( theta ), ct = cos( theta );
```

```
  double sp = sin( phi ), cp = cos( phi );
  double px = e * st * cp;
  double py = e * st * sp;
  double pz = e * ct;
  return HepLorentzVector( px, py, pz, e );
}
```

JetCostituent.h

```
fndef JetConstituent_h
#define JetConstituent_h

#include "Jets/PersistentJet/interface/JetConstituentBase.h"
#include "CARF/Reco/interface/TRecRef.h"

template<typename T>
class JetConstituent : public JetConstituentBase
{
 public:
  // default constructor requires partial specialization
  JetConstituent( const T & );
  JetConstituent() : JetConstituentBase() {}
  ~JetConstituent() {}
  const T * getConstituent() { return ref.get(); }

 private:
  TRecRef<T> ref;
};

#endif
```

JetBuilderBase.h

```
#ifndef JetBuilderBase_h
#define JetBuilderBase_h

#include "CARF/Reco/interface/RecAlgorithm.h"
#include "Jets/PersistentJet/interface/Jet.h"
#include "Utilities/UI/interface/SimpleConfigurable.h"
#include "CARF/Reco/interface/RecCollection.h"

class RecConfig;

template<typename jet>

struct JetBuilderBase : public RecAlgorithm<jet> {
  typedef typename jet::constituent               constituent;
  typedef RecCollection<constituent>              container;
  typedef typename container::const_iterator      const_iterator;

  explicit JetBuilderBase( const RecConfig& cfg ) : RecAlgorithm< jet > ( cfg ) {
    collection = new container( component( "constituents" ) );
  }

  ~JetBuilderBase() { delete collection; }


protected:
  container  *collection;

  typedef std::set<const constituent*> used_set;
  used_set   used_;

  void use( const constituent* c ) { used_.insert( c ); }

  void reUse( const constituent* c )
  {
```

```
    typename used_set::iterator i = used_.find( c );
    if ( i != used_.end() ) used_.erase( i );
}

  bool used( const constituent* c ) const { return used_.find( c ) != used_.end();
  bool unused( const constituent* c ) const { return ! used( c ); }
};

#endif
```

JetBuilder_IterativeCone.h

```
#ifndef JetBuilder_IterativeCone_h
#define JetBuilder_IterativeCone_h

#include "CARF/Reco/interface/RecConfig.h"
#include "CARF/Reco/interface/RecCollection.h"
#include "CARF/Reco/interface/Reconstructor.h"
#include "CARF/Reco/interface/ParameterSetBuilder.h"
#include "CARF/Reco/interface/ComponentSetBuilder.h"
#include "CARF/Reco/interface/RecQuery.h"
#include "Jets/JetAlgos/interface/IterativeConeProtoJet.h"
#include "Jets/PersistentJet/interface/Jet.h"
#include "Jets/PersistentJetAlgos/interface/JetBuilderBase.h"
#include <cmath>
#include <iostream>

template<typename jet>
struct JetBuilder_IterativeCone : public JetBuilderBase<jet> {
  typedef typename RecAlgorithm<jet>::Name               Name;
  typedef typename RecAlgorithm<jet>::Version            Version;
  typedef typename jet::constituent                      constituent;
  typedef typename JetBuilderBase<jet>::const_iterator   const_iterator;
  typedef IterativeConeProtoJet<constituent>             proto;
  static const RecConfig& defaultConfig();
```

```
  explicit JetBuilder_IterativeCone( const RecConfig & cfg = defaultConfig() );
  ~JetBuilder_IterativeCone();
  Name name() const { return defaultConfig().name(); }
  Version version() const { return defaultConfig().version(); }

private:
  void reconstruct();
  double cut, seedEtCut;
};

template<typename jet>
const RecConfig& JetBuilder_IterativeCone<jet>::defaultConfig() {
  static Name name( "Jet_IterativeCone" );
  static Version version( "1.0" );
  ParameterSetBuilder pars;
  pars.addParameter( "cut", 0.7, 0.01 );
  pars.addParameter( "seedEtCut", 0.0, 0.01 );
  ComponentSetBuilder cmp;
  cmp.addComponent( "constituents", RecQuery( "JetConstituent_RawRecParticle" ) );
  static RecConfig cfg( name, version, pars.result(), cmp.result() );
  return cfg;
}

template<typename jet>
JetBuilder_IterativeCone<jet>::JetBuilder_IterativeCone( const RecConfig& cfg ) :
  JetBuilderBase< jet > ( cfg ) {
  cout << "building JetBuilder" << endl;
  setMeAsDefault();
  // doesn't compile:
  // cut_ = parameter<double>( "cut" );
  // seedEtCut_ = parameter<double>( "seedEtCut" );
  const MultiTypeSet& pset = *theParameters;
  cut = pset.MultiTypeSet::value<double>( "cut" );
  seedEtCut = pset.MultiTypeSet::value<double>( "seetEtCut" );
```

```
}

template<typename jet>
JetBuilder_IterativeCone<jet>::~JetBuilder_IterativeCone() { }

template<typename jet>
void JetBuilder_IterativeCone<jet>::reconstruct() {
  int size = collection->size();
  if ( size == 0 ) return;
  if ( size == 1 ) {
    jet *j = new jet;
    j->addConstituent( * ( collection->begin() ) );
    reconstructor_->addObj( j );
    return;
  }

  const_iterator i = collection->begin();
  const constituent  *max = *i;
  int allused = 0;
  double maxEta, maxPhi, maxTEnergy;
  maxEta = max->getEta();
  maxPhi = max->getPhi();
  maxTEnergy = max->getTransverseEnergy();

  if( max != 0 ) do {
     // find max transverse energy constituant
    for ( i = collection->begin(); i != collection->end(); i++ )
      if( unused( *i ) && (*i)->getTransverseEnergy() > maxTEnergy ) {
         max = *i;
         maxEta = max->getEta();
         maxPhi = max->getPhi();
         maxTEnergy = max->getTransverseEnergy();
      }
```

```
    // do the seed-cut; =0 by default.
    if( maxTEnergy < seedEtCut ) {
      max = 0;
      allused = 1;
    }

    if( max != 0 ) {
       // build cone arround it
      proto  *aJet = new proto;
      // ?? aJet->getConstituants();
      use( max );
      aJet->addConstituant( max );
      double deltaEta, deltaPhi, deltaR;
      int iterationNumber = 0;

      do {
iterationNumber++;
for ( i = collection->begin(); i != collection->end(); ++i ) {
          if( unused( *i ) ) {
            deltaEta = (*i)->getEta() - maxEta;
            double phi1, phi2;
            phi1 = (*i)->getPhi();
            if( phi1 < maxPhi ) phi2 = maxPhi;
            else { phi2 = phi1; phi1 = maxPhi; }
            deltaPhi = phi2 - phi1;
            if( deltaPhi > M_PI ) deltaPhi = 2 * M_PI - phi2 + phi1;
            deltaR = sqrt( deltaEta * deltaEta + deltaPhi * deltaPhi );
            if( deltaR < cut ) {
              use(*i);
              aJet->addConstituant(*i);
            }
          }
}
        maxEta = aJet->getEta();
```

```
        maxPhi = aJet->getPhi();
        maxTEnergy = aJet->getTransverseEnergy();
//
if ( ! aJet->done() ) {
  for( typename proto::const_iterator j = aJet->begin();
       j != aJet->end(); ++j )
    reUse( *j );
  aJet->reset();
}
      } while( iterationNumber < 100 && ! aJet->done() );

      if( ! allused ) {
         // fill result
        jet *aFinalJet = new jet;
        for( const_iterator c = aJet->begin(); c != aJet->end(); c++ )
          aFinalJet->addConstituent( *c );
delete aJet;
        reconstructor_->addObj( aFinalJet );
        maxTEnergy = 0;
        max = 0;
      }
    } else { allused = 1; }
  } while( ! allused );
}

#endif
```

JetBuilder_SimpleCone.h

```
#ifndef JetBuilder_SimpleCone_h
#define JetBuilder_SimpleCone_h

#include "CARF/Reco/interface/RecConfig.h"
#include "CARF/Reco/interface/RecCollection.h"
#include "CARF/Reco/interface/Reconstructor.h"
```

```
#include "CARF/Reco/interface/ParameterSetBuilder.h"
#include "CARF/Reco/interface/ComponentSetBuilder.h"
#include "CARF/Reco/interface/RecQuery.h"
#include "Jets/PersistentJet/interface/Jet.h"
#include "Jets/PersistentJetAlgos/interface/JetBuilderBase.h"
#include <cmath>
#include <iostream>

template<typename jet>
struct JetBuilder_SimpleCone : public JetBuilderBase<jet> {
  typedef typename RecAlgorithm<jet>::Name Name;
  typedef typename RecAlgorithm<jet>::Version Version;
  typedef typename jet::constituent constituent;
  typedef typename JetBuilderBase<jet>::const_iterator const_iterator;
  static const RecConfig& defaultConfig();
  explicit JetBuilder_SimpleCone( const RecConfig & cfg = defaultConfig() );
  ~JetBuilder_SimpleCone();
  Name name() const { return defaultConfig().name(); }
  Version version() const { return defaultConfig().version(); }

private:
  void reconstruct();
  double cut, seedEtCut;
  int recom;
};

template<typename jet>
const RecConfig& JetBuilder_SimpleCone<jet>::defaultConfig() {
  static Name name( "Jet_SimpleCone" );
  static Version version( "1.0" );
  ParameterSetBuilder pars;
  pars.addParameter( "cut", 0.7, 0.01 );
  pars.addParameter( "seedEtCut", 0.0, 0.01 );
  pars.addParameter( "recom", 0 );
```

```
  ComponentSetBuilder cmp;
  cmp.addComponent( "constituents", RecQuery( "JetConstituent_RawRecParticle" ) );
  static RecConfig cfg( name, version, pars.result(), cmp.result() );
  return cfg;
}

template<typename jet>
JetBuilder_SimpleCone<jet>::JetBuilder_SimpleCone( const RecConfig& cfg ) :
  JetBuilderBase< jet > ( cfg ) {
  cout << "building JetBuilder" << endl;
  setMeAsDefault();
  // doesn't compile:
  // cut_ = parameter<double>( "cut" );
  // seedEtCut_ = parameter<double>( "seedEtCut" );
  const MultiTypeSet& pset = *theParameters;
  cut = pset.MultiTypeSet::value<double>( "cut" );
  seedEtCut = pset.MultiTypeSet::value<double>( "seetEtCut" );
  recom = pset.MultiTypeSet::value<int>( "recom" );
}

template<typename jet>
JetBuilder_SimpleCone<jet>::~JetBuilder_SimpleCone() { }

template<typename jet>
void JetBuilder_SimpleCone<jet>::reconstruct() {
  int size = collection->size();
  if ( size == 0 ) return;
  if ( size == 1 ) {
    jet * j = new jet;
    j->addConstituent( * ( collection->begin() ) );
    reconstructor_->addObj( j );
    return;
  }
```

```
  const_iterator i = collection->begin();
  const constituent * max = *i;

  int allused = 0;
  double maxEta, maxPhi, maxTEnergy;

  maxEta = max->getEta();
  maxPhi = max->getPhi();
  maxTEnergy = max->getTransverseEnergy();

  if( max != 0 ) do {
      // find max transverse energy constituant
    for ( i = collection->begin(); i != collection->end(); ++i )
      if( unused(*i) && (*i)->getTransverseEnergy() > maxTEnergy ) {
max = *i;
maxEta = max->getEta();
maxPhi = max->getPhi();
maxTEnergy = max->getTransverseEnergy();
      }

      // check seed cut.
      if( maxTEnergy < seedEtCut ) max = 0;

      if( max != 0 ) {
double deltaEta, deltaPhi, deltaR, phi1, phi2;
use( max );
jet  *aJet = new jet;
aJet->addConstituent( max );
for ( i = collection->begin(); i != collection->end(); ++i ) {
  if( unused(*i) ) {
    deltaEta = (*i)->getEta() - maxEta;
    phi1 = (*i)->getPhi();
    if(phi1<maxPhi) phi2 = maxPhi;
    else { phi2 = phi1; phi1 = maxPhi; }
```

```
    deltaPhi = phi2 - phi1;
    if( deltaPhi > M_PI ) deltaPhi = 2 * M_PI - phi2 + phi1;
    deltaR = sqrt(deltaEta*deltaEta + deltaPhi*deltaPhi);
    if( deltaR < cut ) {
      use( *i );
      aJet->addConstituent(*i);
    }
  }
}

// will be used for the recombination of the jet constituants

  // recombination of the final state jet
switch (recom) {
case (0) : {
// here's nothing to do the old fashioned recombination by the JetType
     template (ConcreteJet or E_SchemeJet) is used
  reconstructor_->addObj( aJet );
  break;
}
case (1) : {
  HepLorentzVector the4Vector( 0, 0, 0, 0 ) ;

  // E scheme
  if( aJet->size() != 0 )
    // loop over all jet constituents with simple 4-vector addition
    for( typename jet::const_iterator i = aJet->begin(); i != aJet->end(); ++i )
      the4Vector += (*i)->getLorentzVector();

  // addinf the recombinated jet
  reconstructor_->addObj( new jet( the4Vector ) );
  // deleting of the not used jet
  delete aJet;
```

```
  break;
}
case (4) : {
  HepLorentzVector the4Vector( 0, 0, 0, 0 ) ;

  // Et scheme
  if( aJet->size() != 0 ) {
  // loop over all jet constituents and apply the Et recombination scheme
   for(typename jet::const_iterator i = aJet->begin(); i != aJet->end(); ++i){
      HepLorentzVector a4Vector;
      double anEnergy = (*i)->getEnergy();
      double aPhi =     (*i)->getPhi();
      double aTheta =   (*i)->getTheta();
      double st = sin( aTheta ), ct = cos( aTheta );
      double sp = sin( aPhi ), cp = cos( aPhi );
      a4Vector.setX( anEnergy * st * cp );
      a4Vector.setY( anEnergy * st * sp );
      a4Vector.setZ( anEnergy * ct );
      a4Vector.setT( anEnergy * st );
      the4Vector += a4Vector;
    }
    the4Vector.setT(the4Vector.t()/sin(the4Vector.theta()));
  }

  // addinf the recombinated jet
  reconstructor_->addObj( new jet( the4Vector ) );

  // deleting of the not used jet
  delete aJet;

  break;
}
default :
  // todo
```

```
  break;
}

maxTEnergy = 0;
max = 0;
      }
      else { allused = 1; }
  } while(!allused);
}

#endif
```

JetCostituentBuilder.h

```
#ifndef JetConstituentBuilder_h
#define JetConstituentBuilder_h

#include "Jets/PersistentJetAlgos/interface/RecConverter.h"
#include "Jets/PersistentJet/interface/JetConstituent.h"

template<typename T>

struct JetConstituentBuilder : public RecConverter< T, JetConstituent<T> > {
  typedef JetConstituent<T> constituent;
  typedef typename RecAlgorithm<constituent>::Name Name;
  typedef typename RecAlgorithm<constituent>::Version Version;


  explicit JetConstituentBuilder( const RecConfig& cfg = defaultConfig() ) :
    RecConverter< T, constituent > ( cfg, "components" ) { }


  Name name() const { return defaultConfig().name(); }
  Version version() const { return defaultConfig().version(); }
```

```
  static const RecConfig & defaultConfig();
};

#endif
```

Elenco delle figure

2.1 Collaboration 5
2.2 Anello 7
2.3 Collisione particelle 8
2.4 CMS 9
2.5 SistRif 10
2.6 I rivelatori dell'apparato CMS 12
2.7 Il rivelatore a pixel di silicio. 13
2.8 Magnete 15
2.9 Muon Detector 16
2.10 Il tipo di particelle riconosciute nei vari sottorivelatori 19
2.11 foto 20
2.12 foto2 21

3.1 Particelle e sottorivelatori 23
3.2 CMS Software Project 26
3.3 Rilevamenti nei caolorimetri. 29
3.4 Diagramma di flusso delle fasi dell'esperimento 30
3.5 Observer 33
3.6 Modello Dati 35
3.7 Classi della Ricostruzione 38
3.8 Coni 40
3.9 Algoritmo di tipo JADE. 42

4.1 UML dell'architettura dei *jet* corrente 46
4.2 Il framework di ricostruzione dei *jet* (transiente) 48

4.3 Diagramma UML per la rappresentazione dei *jet* persistente (I versione). 50
4.4 Diagramma UML per la rappresentazione dei *jet* persistente (II versione). 52
4.5 Framework per la ricostruzione persistente (II Versione) 53
4.6 Framework per la ricostruzione persistente (III Versione) . . . 55

Bibliografia

[1] CMS, The Compact Muon Solenoid: technical proposal CERN-LHCC - 54 - 38, CERN-LHCC-p-1, Diction 1994.

[2] LHC Technical Design Report 1994

[3] CERN http://www.cern.ch

[4] ALICE Technical proposal, CERN-LHCC-95-71, CERN-LHCC-P-3, Dic. 1995

[5] LHCB Technical proposal, CERN-LHCC-58-4, CERN-LHCC-P-4.

[6] LEPII Search for the Standard Model Higgs Boson at LEP, AEDH, DELPHI, L3, OPAL

LEP Worky Gray for Higgs boson search, K. Barate et al., Phys. Lett. B565:61-72, 2003

[7] ORCA http://cmsdoc.cern.ch/orca/

[8] QCD D. Peskin, Introduction to High Energy Physic, Cambrige, 2000

[9] Stroustrup, Bjarne. The C++ Programming Language, 3rd Edition, Addison-Wesley, 1997. ISBN 0-201-88954-4.

[10] OSCAR http://cmsdoc.cern.ch/oscar/

[11] P.W. Higgs, Phys Lett. 12 (1964) 132;

P.W. Higgs, Phys Rev. Lett. 13 (1964) 508;

P.W. Higgs, Phys Rev. 145 (1966) 1156;

F. Euglert, R Brout, Phys. Rev. Lett. 19 (1967) 321;

G. S. Guralnik, C.R.Hagen, T.W.B. Kibble, Phys. Rev. Lett. 13 (1964) 585.

[12] S.Ferrera, Supersimmetry (Ansherdam, North Hollud, World Scintle), 1987, vol. 1 e 2;

J. Boggin, 1983, Supersimmetry and SuperGravity (Princeton HP).

[13] UML http://www.omg.org/uml/

[14] ROOT http://root.cern.ch

[15] http://wwwps.lnf.infn.it/particle/paitaliano/map_proj.html

[16] http://www-collider.physics.ucla.edu/cms/cmsim/cmsim/manual/www/cms113/manual.html

[17] PYTHIA http://www.thep.lu.se/ torbjorn/Pythia.html

[18] POOL http://pool.cern.ch/PoolUserGuide.pdf

[19] S. Weinberg, Phys Rev. Lett. 19 (1967) 1264;
A. Salon, Elementary Particle Theory, ed. N. Svortholn, 1968, 367

[20] ATLAS Technical proposal CERN-LHCC - 94 - 43, CERN-LHCC-P-2, Dic. 1994.

[21] GEANT4 http://wwwinfo.cern.ch/asd/geant4/geant4.html

[22] Design Pattern; Gamma, Helm, Johnson, Vlissides; Addison-Wesley; 2002